

Юхопер

Введение в логическое программирование



Издательство «Мир»

Введение в логическое программирование

**Introduction
to logic
programming**

Ghristopher John Hogger

**Department of Computing
Imperial College of Science and Technology
London, United Kingdom**

1984

Academic Press, Inc.

**Harcourt Brace Jovanovich, Publishers
London Orlando San Diego
New York Austin Montreal Sydney
Tokyo Toronto**

К.Хоггер Введение в логическое программирование

Перевод с английского
М. В. Захарьящева

под редакцией
Ю. И. Янова



Москва «Мир», 1988

ББК 32.973

Х68

УДК 681.3

Хоггер К.

Х 68 Введение в логическое программирование: Пер. с англ. — М.: Мир, 1988. — 348 с., ил.

ISBN 5-03-000490-4

Книга написана английским специалистом по программированию и знакомит читателей с фундаментальными идеями и методологией логического программирования. В ней много внимания уделено вопросам синтеза программ, реализации языков логического программирования и их применения. Материал тщательно отобран, приводится много примеров, облегчающих усвоение материала.

Для математиков-прикладников, специалистов по информатике, программистов, аспирантов и студентов.

X $\frac{1702070000-406}{041(01)-88}$ 46—88, ч. I

ББК 32.973

Редакция литературы по математическим наукам

ISBN 5-03-000490-4 (русск.)

ISBN 0-12-352090-8 (англ.)

© 1984, by Academic Press
Inc. (London) Ltd.

© перевод на русский язык,
«Мир», 1988

Предисловие редактора перевода

Основная трудность программирования задач для ЭВМ связана не столько с различием языков человека и машины, сколько с различием их «мышления». Долгое время считалось, что машина способна правильно воспринимать и исполнять только алгоритмы, т. е. детерминированные формальные исчисления директивного типа. В то же время первоначальная формулировка задачи имеет, как правило, описательный характер. Неформальный этап построения подходящего алгоритма и запись его на определенном формальном языке требует от программиста не только значительных затрат времени, но и достаточно высокой квалификации. Этот и некоторые другие недостатки алгоритмического подхода стимулировали поиск иных возможностей. Осознание того, что вычисление есть частный случай логического вывода, а алгоритм — это аксиоматическое задание функции, привело к идеи так называемого логического программирования, первая реализация которой была осуществлена в начале 70-х годов в виде системы Пролог. Суть этой идеи состоит в том, что машине в качестве программы можно предоставить не алгоритм, а формальное описание предметной области и задачи (функции) в виде аксиоматической системы, и тогда построение решения задачи в виде вывода в этой системе можно поручить самой машине. Таким образом, от программиста уже не требуется построения алгоритма, решающего задачу, поскольку нужный алгоритм порождается интерпретатором, строящим вывод по определенной стратегии. Теперь основная задача программиста — удачно аксиоматизировать, т. е. описать в виде системы логических формул предметную область и такое множество отношений на ней, которые с достаточной степенью полноты описывают задачу. Следует сразу отметить, что этот подход был бы нереален, если бы не существовало методов автоматического поиска доказательств.

Практическая эффективность логического программирования зависит не только от удачной аксиоматизации задачи, но и в еще большей мере от качества интерпретатора, реализующего поиск доказательства. Поэтому кроме техники программирования очень важно совершенствовать методы автоматического доказа-

тельства теорем. Но независимо от того, каков будет незамедлительный прикладной эффект, сама идея логического программирования открывает новые возможности использования ЭВМ. Об этом же свидетельствует и запланированное использование Пролога в машинах пятого поколения.

Предлагаемая читателям книга значительно отличается от немногочисленной пока литературы на русском языке, посвященной логическому программированию, в частности Прологу. Она объединяет в себе детальное и в то же время популярное изложение теоретических основ нового направления, широкий исторический обзор различных аспектов вопроса с практическим руководством по обучению искусству программирования на Прологе. Благодаря этому книгу можно рекомендовать читателям с разнообразными запросами — от профессиональных программистов до теоретиков, интересующихся прикладными вопросами математической логики и теории вычислимости.

Ю. И. Янов

Предисловие

Эта книга представляет собой значительный вклад в логическое программирование. Впервые на страницах одной книги дается всестороннее и тем не менее доступное введение во все аспекты данного предмета. Она занимает промежуточное положение между практическими введениями в Пролог Клоксина и Меллиша [Клоксин, Меллиш (1981)], а также Кларка и Маккейба [Кларк, Маккейб (1982)], с одной стороны, и более общими подходами к вычислительной логике, изложенными, например, Робинсоном и мною, — с другой.

В ней охватываются два важных аспекта логического программирования, которые нигде больше не рассматривались, а именно: вывод логических программ из логических спецификаций и реализация Пролога. Первый из них является большим вкладом логического программирования в решение классических проблем разработки программного обеспечения, и здесь сам автор был инициатором исследований и сыграл в них значительную роль. Второй аспект представляет большой теоретический и коммерческий интерес, и многие приверженцы Пролога будут благодарны за доступное изложение этого предмета, известного до сих пор лишь посвященным.

Эта превосходно написанная книга доставит удовольствие как начинающим, так и специалистам. Начинающим она поможет войти в более широкий мир логического программирования, который простирается за рамки Пролога, а специалистам по логическому программированию — глубже понять предмет и с еще большим энтузиазмом вести исследования.

Имперский колледж, Лондон
май 1984

Роберт Ковальский

Предисловие автора

Многие ожидают, что символическая логика будет служить в качестве основной формальной системы программирования для следующего поколения ЭВМ. Выбор логики на эту роль в японском проекте создания компьютеров пятого поколения вызвал во всем мире широкий интерес к логическому программированию, хотя и до того становилось ясно, что этой очаровательной формальной системе суждено было сделать значительный вклад в теорию и практику вычислений.

Развитием логического программирования до недавнего роста интереса к нему занимались лишь несколько исследовательских центров, в результате чего существующая литература по этому предмету еще довольно мала. Те немногие тексты, которые уже опубликованы, основаны на специфических реализациях языка логического программирования, другие вот-вот появятся, но в большей своей части эти книги предназначены быть учебными руководствами по написанию программ. Кроме того, для специалистов в области информатики были изданы в виде книг сборники статей о последних достижениях в логическом программировании. Остается, таким образом, довольно большой пробел между этими двумя крайностями в уровне изложения, и главная цель данной книги — до некоторой степени его восполнить.

В первой половине книги логическое программирование вводится на уровне учебного пособия, однако в дополнение здесь дается гораздо больше теоретического и исторического материала, чем обычно можно было бы ожидать в учебнике по программированию. Уровень изложения соответствует курсу информатики, читаемому студентам первого года обучения. Во второй половине рассматриваются более сложные аспекты логики как вычислительного формализма. Цель этой части книги — собрать воедино, упростить и объяснить избранные темы из подчас разрозненной и технически очень сложной исследовательской литературы, а также дать обзор последних достижений в теории и приложениях. Она может быть использована в качестве справочника как студентами, специализирующимися в области логического программирования, так и теми исследователями, для которых эта область является сравнительно новой.

Своей попыткой написать эту книгу я в большой степени обязан чести работать вместе с другими исследователями в области логического программирования в отделении информатики Имперского колледжа. В особенности я благодарен Р. Ковальскому и К. Кларку, чья поддержка помогла мне проявить настойчивость в завершении работы. Я признателен также М. ван Эмдену и Дж. Ллойду, которые прочитали большую часть рукописи и сделали свои замечания. Суровое требование — оставаться на уровне самых последних исследовательских работ в данной области — значительно упростило великолепный Информационный бюллетень по логическому программированию, выходящий в лиссабонском университете под ред. Л. М. Перейры, и я уверен, что мою высокую оценку его работы разделяют также и все другие специалисты по логическому программированию.

Значения основных символов

Символ	Значение
,	(запятая) и
∨	или
¬	не
↔	если
↔↔	тогда и только тогда, когда
↔↔↔	тогда и только тогда, когда
∀	для всех
∃	для некоторого (существует)
:=	присвоить значение
⊨	логически следует
□	доказуемо
~	недоказуемо
—	такой что
□	успех (путем доказательства от противного)
■	неудача
≠	не равно
⊕	векторная сумма
△	если доказуемо, то ввести новый факт (породить лемму)
∈	принадлежит (является элементом)
⊆	является подмножеством
∪	объединение множеств
∅	пустое множество

Введение

Тема этой книги — использование символьической логики как языка программирования. К моменту ее написания логика в этом качестве применялась не более двенадцати лет, и подробности все еще не известны большей части программистского сообщества. Ситуация, по-видимому, быстро изменится благодаря тому, что логическое программирование было не так давно признано ключевым формализмом для следующего поколения компьютеров.

Существенное отличие логического программирования от традиционного заключается в том, что мы должны описывать логическую структуру задач, а не указывать компьютеру, как именно ему следует их решать. Люди, не имеющие предыдущего опыта вычислений на ЭВМ, склонны считать, что программирование есть и всегда было по природе своей логическим делом, и часто они бывают удивлены или даже разочарованы, когда, познакомившись с языками, подобными языку Бейсик, обнаруживают, что на самом деле это не так. Они узнают вместо этого, что традиционный способ написания программ сильно зависит от внутренних механизмов компьютера, что, конечно, само по себе разумно, но тем не менее не связано, по-видимому, непосредственно с исходным пониманием задачи. Наоборот, программисты, обученные использованию только традиционных языков, при знакомстве с логическим программированием могут столкнуться со сравнимыми трудностями приспособления. Инстинктивно, желая эффективно управлять машиной, они испытывают смутное чувство потери, когда осваивают язык, не имеющий машинно-ориентированной специфики. У них может наблюдаться программистский эквивалент синдрома лишения, следующий за длительным периодом наркотического пристрастия к оператору присваивания.

Приспособиться к языку, как мне известно из собственного опыта, не всегда оказывается легко. До сих пор хорошо помню, как, слушая студентом вводный курс языка Фортран, я в состоянии был понять описание воздействия на машину отдельных операторов, но не был уверен в том, как их следует связать вместе в соответствии с логической структурой задачи. Помнится

также, как несколько лет спустя уже преподавателем Фортрана я впервые увидел утверждение языка логического программирования, и мне было непонятно, каким образом оно может способствовать алгоритмическому решению задачи на машине. Нужно приложить определенные усилия для того, чтобы преодолеть длительную привычку к одному-единственному представлению о вычислениях.

В первом утверждении языка логического программирования, которое мне показали, говорилось: «Вы здоровы, если вы едите овсянную кашу». Это предложение повседневного языка становится предложением языка символьической логики, если его структурировать следующим образом:

здоров (*u*) если есть (*u*, КАША)

В этой записи представлены все основные компоненты того языка, который мы будем использовать для целей программирования: индивидуальные объекты (КАША), переменные (*u*), стоящие вместо каких-либо объектов, высказывания об объектах, такие как здоров (*u*) и есть (*u*, КАША), а также связи (если), соединяющие высказывания. Приведенное выше предложение могло бы являться частью «экспертной системы», предлагающей консультацию (в данном примере сомнительной ценности) по вопросам личного питания. Столь же легко мы можем сформулировать какое-либо утверждение в более математическом духе, например:

четное-число (*u*) если делится (*u*, 2)

Как же использовать подобные логические описания для того, чтобы заставить компьютер решить задачу? Рассмотрим следующую аналогию. Вы желаете совершить путешествие на автомобиле, определив начало, конец и, возможно, какие-то еще детали своего маршрута. Для прохождения этого маршрута потребуется принимать разнообразные решения, зачастую незначительные и повторяющиеся, относительно управления автомобилем и соблюдения правил дорожного движения. Теперь вам предлагается совершить запланированное путешествие, не принимая ни одного из этих решений. Как это сделать? Поручить какому-то другому водителю вести машину и сообщить ему все ваши требования к маршруту.

В логическом программировании таким «водителем» является логический интерпретатор — программа, знающая, как использовать компьютер для того, чтобы выводить следствия из произвольного множества логических предложений. Программист должен обеспечить как правильность данных предложений, так и достаточную их информативность для того, чтобы требуемые следствия оказались выводимыми.

Рассмотрим более конкретный пример. Представим себе, что некоторая часть оборудования контролируется устройством индикации с тремя индикаторными лампами; каждая из ламп либо включена (*ВКЛ*), либо выключена (*ВЫКЛ*). По различным комбинациям состояний *ВКЛ/ВЫКЛ* ламп на этом устройстве оператор периодически определяет, в каком состоянии — *ВКЛ* или *ВЫКЛ* — должен находиться некоторый переключатель на нашем оборудовании. Используя логику, мы можем написать ряд простых фактов

R1 : правило(*ВКЛ*, *ВКЛ*, *ВЫКЛ*, *ВКЛ*)
R2 : правило(*ВЫКЛ*, *ВКЛ*, *ВКЛ*, *ВЫКЛ*)
 .
 .
 .

и т. д.,

где каждое высказывание правило (w, x, y, z) означает, что переключатель следует поставить в состояние w , когда устройство индикации находится в состоянии x, y, z . Все вместе эти предложения можно использовать в качестве таблицы решений. Пусть у нас возникло желание хранить эту таблицу в памяти компьютера так, чтобы оператор мог спрашивать у него, какое состояние w соответствует показываемому устройством индикации состоянию x, y, z . Если на компьютере реализован логический интерпретатор, то оператору нужно только спросить, является ли некоторое конкретное высказывание следствием предложений, хранящихся в таблице. Он может, например, ввести в машину логический запрос

? правило($w, ВКЛ, ВЫКЛ, ВКЛ$)

который спрашивает, при каких значениях w предложение правило($w, ВКЛ, ВЫКЛ, ВКЛ$) является следствием R1, R2 ... и т. д. Согласно R1, таким значением оказывается *ВКЛ*, поэтому интерпретатор сам обнаружит это и выдаст ответ $w := ВКЛ$.

Без изменения хранимых в памяти предложений можно получать ответы и на многие другие запросы, просто предъявляя их интерпретатору. В результате запроса ?^правило(*ВЫКЛ*, x, y, z) будут выданы все состояния x, y, z устройства индикации, требующие ответа *ВЫКЛ*. Запрос ?^правило(*ВЫКЛ*, *ВКЛ*, *ВКЛ*, *ВЫКЛ*) просит только подтвердить, что *ВЫКЛ* является правильным ответом на состояние *ВКЛ, ВКЛ, ВЫКЛ*; интерпретатор (в силу правила R2) ответит просто «ДА». Запрос ?правило (w, x, y, z) вызовет распечатку всей таблицы решений. Запрос ?^правило (w_1, x, y, z), правило (w_2, x, y, z), $w_1 \neq w_2$, спрашивая, не имеет ли какое-либо состояние x, y, z несколько вхождений в таблицу с разными ответами, инициирует проверку

таблицы на противоречивость. Если же мы поместим в память машины еще два предложения

S1 : состояніе(*ВКЛ*)
S2 : состояніе(*ВЫКЛ*)

то запрос ? состояніе (*x*), состояніе (*y*), состояніе (*z*), ~правило (*w*, *x*, *y*, *z*), спрашивая, не пропущены ли в таблице какие-либо состояния *x*, *y*, *z* (~ означает «не»), а если пропущены, то какие, инициирует проверку таблицы на полноту.

Короче говоря, если на вопрос можно логически получить ответ, используя хранимые в памяти предложения, то на него будет дан интерпретатором ответ при помощи логического вывода. Фактически в любом другом языке программирования для каждой новой задачи, которую нужно решать с помощью фиксированного объема знаний, требуется составлять новую программу, и чем сложнее оказывается решение задачи, тем сложнее будет и эта программа. Отсутствие гибкости программ при изменении цели должно значительно снижать продуктивность программирования.

Было бы неверным сейчас создать ложное впечатление о том, что логика освобождает программиста от pragматических соображений. В реальных приложениях с целью достижения приемлемой эффективности исполнения программы часто бывает необходимо структурировать входные предложения, должным образом учитывая дедуктивную стратегию интерпретатора и конкретный вид поставленного запроса. Программисту поэтому требуется, как правило, думать и об алгоритмических и о дескриптивных свойствах создаваемых им программ. Важным моментом тем не менее является то, что утверждения программы (т. е. входные логические предложения) всегда будут логически описывать саму задачу, а не процесс ее решения: точные допущения, сделанные о задаче, будут всегда непосредственно видны из текста программы. На протяжении всей этой книги мы будем уделять особое внимание этому соображению, а также влиянию, которое оно оказывает на методологию программирования и более широкие проблемы, связанные с разработкой программного обеспечения.

Логическому программированию с успехом обучали детей младшего школьного возраста, используя при этом содержательные понятия логического следствия и логического вывода. Такой неформальный подход оказался в высшей степени полезным, поскольку он позволяет неискушенному пользователю сравнительно безболезненно усвоить основные принципы. Однако для правильной оценки исторических и теоретических оснований данного формализма требуется более тщательное рассмотрение. С учетом этого цель первой главы состоит в том,

чтобы дать достаточно точное и полное описание логики как языка решения задач, объясняющее структуру предложений, понятия логического следствия и логического вывода. Вторая глава имеет более вычислительный характер. В ней рассматривается главным образом процедурная интерпретация логики и показывается, как знакомые алгоритмические процессы извлекаются интерпретатором из логических программ. В гл. III и IV приводятся прагматические и стилистические соображения по поводу структурирования программ и данных. Таким образом, главная задача первой половины книги — объяснить, как следует понимать и составлять логические программы.

Вторая половина книги написана преимущественно для специалистов по информатике, и, следовательно, она носит более технический характер. В гл. V и VI обсуждаются спецификация, верификация и синтез программ, а в гл. VII описываются элементарные свойства типичных реализаций языка логического программирования. В последней главе показывается, какой вклад внесло логическое программирование в общую теорию вычислений. В ней приводятся наиболее важные результаты теории логического программирования, описываются некоторые аспекты деятельности, связанной с разработкой интеллектуальных систем, основанных на знании, а также объясняется роль математической логики в будущих компьютерах пятого поколения.

I. Представление знаний и рассуждения

Логическая программа состоит из предложений, выражающих знания о той задаче, для решения которой программа предназначается. Для формулировки этих знаний используются два основных понятия: наличие дискретных объектов, которые здесь называются *индивидуумами*, и наличие *отношений* между ними. Индивидуумы, рассматриваемые в контексте каждой конкретной задачи, образуют все вместе *проблемную область* этой задачи. Если, к примеру, задача состоит в том, чтобы решить алгебраическое уравнение, то проблемная область может состоять из вещественных чисел (или по крайней мере включать их).

Для того чтобы индивидуумы и отношения можно было представлять в такой символической системе, как логика, им следует дать *имена*. Именование — это предварительный этап построения символьических моделей, представляющих то, что нам известно. Главная задача — построить *предложения*, выражающие различные логические свойства именованных отношений. Рассуждения о некоторой задаче, поставленной на данной проблемной области, можно проводить, манипулируя этими предложениями при помощи логического *вывода*. В контексте логического программирования программист обычно придумывает предложения, образующие его программу, а компьютер затем строит необходимый для решения задачи вывод. Чтобы все это делалось эффективно, программист должен быть достаточно квалифицированным как в представлении знаний, так и в понимании того, как они будут обрабатываться на машине. В этой главе мы введем язык *логики первого порядка* и покажем, как его можно использовать в качестве средства для представления знаний и рассуждений, а следовательно, и для решения задач на ЭВМ.

I.1. Индивидуумы

Индивидуумами могут быть совершенно произвольные объекты, например числа, геометрические фигуры, уравнения или программы для ЭВМ. Очень часто достаточно дать им простые имена, такие как

1 2 ЕДИНИЦА ДВОЙКА ОКРУЖНОСТЬ
УРАВНЕНИЕ-1 ПРОГРАММА-2

которые выбираются из некоторого заданного словаря. Эти имена неделимы (или неструктуримы), и обычно их называют *константами*. У каждого конкретного индивидуума, если потребуется, может одновременно быть какое угодно количество имен. Так, *ЕДИНИЦА* и *I* могут быть именами индивидуума, известного как первое положительное целое число. Выбираются имена произвольным образом, и поэтому первому положительному целому числу можно было бы (упорио) давать имя *З*, если уж так очень хочется.

Иногда бывает удобно давать индивидуумам составные (структурированные) имена, такие как

УДВОИТЬ(2) СЛОЖИТЬ(1,2)

Каждое из них состоит из кортежа длины *n*, которому предшествует функтор (или функциональный символ). Кортеж длины *n* — это просто произвольный упорядоченный набор из *n* имен. Так, *(1, 2)* является примером кортежа длины 2. Скобки, в которые заключается кортеж, служат только для обозначения его начала и конца, и при желании они могут быть опущены. Кортеж длины 2 можно называть просто *парой*, а кортеж длины 3 — *тройкой*. Функторы, такие как *УДВОИТЬ* и *СЛОЖИТЬ*, также выбираются произвольным образом из другого заданного словаря. Каждый функтор может предшествовать только кортежем некоторой фиксированной длины *n*, и в этом случае его называют *n*-местным (или *n*-арным) функтором. Так, в рассматриваемом примере *УДВОИТЬ* — это 1-местный (или 1-арий, или униарий) функтор, а *СЛОЖИТЬ* — 2-местный (или 2-арный, или бинарный) функтор.

Функторы дают возможность строить имена какой угодно сложности, например

СЛОЖИТЬ(УДВОИТЬ(2), СЛОЖИТЬ(1, УДВОИТЬ(1))

Это имя, которое можно было бы дать седьмому положительному целому числу [поскольку его можно рассматривать как $2 \cdot 2 + (1 + 2 \cdot 1) = 7$], указывает, что данный индивидуум зависит от двух других индивидуумов с именами *УДВОИТЬ(2)* и *СЛОЖИТЬ(1, УДВОИТЬ(1))* соответственно. Самый внешний функтор *СЛОЖИТЬ* по существу служит именем этой зависимости.

I. 2. Отношения

Символ, подобный *УДВОИТЬ*, не имеет внутреннего, присущего только ему значения, и потому он сам по себе не соответствует нашему интуитивному представлению об умножении на два. В основе этого представления лежит конкретное множество

пар чисел (простоты ради мы ограничимся здесь только натуральными числами), и один из способов его формализации заключается в том, чтобы выбрать для чисел имена $1, 2, 3, \dots$ и т. д. и, объединив их в пары, образовать затем следующее множество, которому присвоено имя **удвоить**

$$\text{удвоить} = \{(1,2), (2,4), (3,6), \dots \text{ и т. д.}\}$$

Вхождение какой-либо пары (x, y) в это множество может означать тогда, что удвоенное число x равно y , а все множество целиком можно рассматривать как полностью охватывающее понятие «**удвоить**».

Данное множество является примером отношения: *n-местное* (или *n-арное*) *отношение* — это любое множество кортежей длины n для некоторого фиксированного значения n . Так, например, **удвоить** есть имя 2-местного (или 2-арного, или бинарного) отношения; оно *связывает индивидуумы*, имена которых входят в каждый принадлежащий этому отношению кортеж. Отметим, что мы не обязаны выбирать кортежи, которые обладают какой-либо внутренней зависимостью. При желании мы можем построить отношение, содержащее пару (*завтрашняя погода*, *высота Эйфелевой башни*).

Каждое понятие представимо бесконечным числом способов. Мы могли бы, например, определить другое, более общее отношение с именем **равно**, содержащее такие пары, как

$$(\text{УДВОИТЬ}(1), 2) \quad (\text{УДВОИТЬ}(2), 4) \quad (\text{УТРОИТЬ}(2), 6)$$

Тогда вхождение какой-либо пары (x, y) в отношение **равно** может означать, что x равно y . В частности, пара $(\text{УДВОИТЬ}(x), y)$, входящая в **равно**, может означать, что удвоенное число x равно y . Поэтому понятие «**удвоить**» представимо множеством всех таких пар, принадлежащих отношению **равно**.

Та степень, в которой отношение представляет какое-либо понятие, зависит не от ассоциаций, вызываемых словами типа «**удвоить**», а скорее от логических свойств самого отношения. Поэтому если мы хотим, чтобы наше отношение **удвоить** верно представляло интуитивный смысл операции умножения на два, то оно не должно содержать пар (x, x) ни для каких конкретных значений x . Это только одно из многих вытекающих из законов целочисленной арифметики свойств, которому должно удовлетворять данное отношение, чтобы выполнять предназначенную ему роль.

Некоторые факты можно представлять, пользуясь лишь кортежами длины 1. Например, свойство «быть целым отрицательным числом» можно было бы представить множеством

$$\{-1, -2, -3, \dots \text{ и т. д.}\}$$

Такие множества, строго говоря, называются *свойствами*, а не *1-местными отношениями*. Поскольку, однако, может быть утомительным все время проводить различия между свойствами и отношениями, в дальнейшем можно считать, что все рассуждения, касающиеся отношений, касаются также и свойств.

I.3. Предикаты, связки и формулы

В логике отношениям даются имена — *предикатные символы*, которые выбираются из заданного словаря. Знания об отношениях выражаются тогда предложениями, построенными из предикатов, связок и формул. Каждый *n-местный* (или *n-арный*) предикат образуется из кортежа длины *n*, перед которым ставится *n-местный* (или *n-арный*) предикатный символ. Предикатом будет, например, *удвоить* (2, 4) где *удвоить* есть 2-местный предикатный символ. Этот предикат читается как высказывание о том, что пара (2, 4) принадлежит отношению с именем *удвоить*. Мы называем здесь 2 и 4 соответственно первым и вторым аргументами предиката *удвоить* (2, 4). В более общем случае предикат *p(t)*, где *t* — некоторый кортеж длины *n*, можно неформально читать либо как «*t* принадлежит отношению *p*», либо как «высказывание *p* справедливо для *t*».

В логике имеются, кроме того, несколько *связок*:

, ∨ ⊥ ← ↔

которые читаются как «и», «или», «не», «если» и «тогда и только тогда, когда» соответственно. Связки используются для построения формул, соединяя предикаты или какие-то другие формулы.

Формулы простого вида определяются следующими правилами:

- (i) каждый предикат есть формула;
- (ii) если *F1* и *F2* — формулы, то формулами являются также

$$(F1) \quad F1, F2 \quad F1 \leftarrow F2$$

$$\neg F1 \quad F1 \vee F2 \quad F1 \leftrightarrow F2$$

В дальнейшем мы расширим определение формулы, допуская вхождения переменных в аргументы предикатов. Правилами (i) и (ii) предусматривается построение только *формул без переменных*.

Заметим, что в правиле (ii) вводятся *знаки пунктуации*: (и). Таким образом, мы можем отличать, например, формулу

$$F1 \leftarrow (F2, F3) \text{ от формулы } (F1 \leftarrow F2), F3$$

а формулу

$$\neg(F2, F3) \text{ от формулы } (\neg F2), F3$$

Принимая соглашение о том, что \neg связывает сильнее, чем остальные связки, а $,$ и \vee — сильнее, чем \leftarrow и \leftrightarrow , мы можем опускать некоторые вхождения знаков пунктуации, и при этом двусмыслистности не возникает. Мы можем писать

$F1 \leftarrow F2, F3$ вместо $F1 \leftarrow (F2, F3)$

и

$\neg F2, F3$ вместо $(\neg F2), F3$

Кроме того, разрешается писать

$F1, F2, F3$ вместо $(F1, F2), F3$ или вместо $F1, (F2), F3$

и

$F1 \vee F2 \vee F3$ вместо $(F1 \vee F2) \vee F3$ или вместо $F1 \vee (F2 \vee F3)$

В логическом программировании наиболее часто используются связки «и», «не» и «если». Как правило, они применяются для построения формул вида

**положительное(6) \leftarrow отрицательное(-2),
отрицательное(-3), умножить(-2, -3, 6)**

Эта формула является примером формулы без переменных, и ее можно читать как

**6 — положительное число если -2 — отрицательное число и
 -3 — отрицательное число и
произведение чисел -2 и
 -3 равно 6**

Данное предложение — средней сложности, а наиболее простые предложения состоят из одного предиката или предиката с отрицанием, например:

отрицательное(-1) и \neg положительное(-1)

Общая структура предложений будет описана после обсуждения переменных.

I.4. Переменные

Очень часто приходится делать утверждения о «всех» индивидуумах. Точное значение понятия «все» будет определено ниже, в разд. I.7, где рассматриваются интерпретации предложений. В качестве примера подобного утверждения можно привести следующее:

**для всех x и y , (*УДВОИТЬ*(x), y) принадлежит
отношению равно если (x , y)
принадлежит отношению удвоить**

Один из способов выражения этого утверждения на языке логики заключается в том, чтобы, выписывая предложения без переменных, рассмотреть «все» имеющиеся в виду значения x и y , например:

```
равно(УДВОИТЬ(1), 2) ← удвоить(1, 2)
равно(УДВОИТЬ(2), 4) ← удвоить(2, 4)
.
.
и т. д.
```

Этот метод, очевидно, слишком громоздкий, чтобы быть практически применимым. Использование переменных позволяет вместо приведенных выше предложений написать одно обобщенное предложение, а именно

$$\text{равно}(\text{УДВОИТЬ}(x), y) \leftarrow \text{удвоить}(x, y)$$

где x и y выбираются из некоторого заданного словаря *переменных*. Таким образом, разрешая ставить переменные вместо конкретных имен, мы приходим к более общим понятиям кортежа длины n , формулы и предложения. Чтобы отличать переменные от констант, в этой книге мы принимаем соглашение: все переменные начинаются со строчных букв, как, например,

$$x \ y \ i \ j \ alpha \ abc \ 123$$

Роль переменной в предложении определяется способом ее *квантификации*, которая может быть либо *универсальной*, либо *экзистенциальной*. В рассмотренном выше примере имеется в виду универсальная квантификация. Ее можно указать явно с помощью символа \forall , который читается «для всех». Таким образом, чтобы сделать подразумеваемое явным, мы должны записать это предложение в виде

$$(\forall x \forall y) (\text{равно}(\text{УДВОИТЬ}(x), y) \leftarrow \text{удвоить}(x, y))$$

Экзистенциальная квантификация употребляется тогда, когда требуется указать на «некоторый» (по крайней мере один) индивидуум. С этой целью используется символ \exists , который читается «существует». Рассмотрим, например, следующее простое определение из теории множеств

для всех y , y есть непустое множество тогда и только тогда, когда существует x , такой что x принадлежит y

В логике это определение можно выразить предложением

$$(\forall y) (\text{не-пусто}(y) \leftrightarrow (\exists x) \text{ принадлежит}(x, y))$$

Выражения $(\forall y)$ и $(\exists x)$ называются соответственно *квантором всеобщности* и *квантором существования*. Областью действия каждого квантора является формула, перед которой он стоит. Так, в предыдущем предложении областью действия квантора $(\forall y)$ будет формула

$$(\text{не-пусто}(y) \leftrightarrow (\exists x) \text{ принадлежит}(x,y))$$

а областью действия квантора $(\exists x)$ будет формула *принадлежит* (x,y) .

I.5. Предложения

Имея переменные, мы можем расширить теперь определение предиката и формулы. В соответствии со следующими определениями предикаты могут содержать теперь в качестве своих аргументов термы, а не только конкретные имена:

(i) *термом* является либо константа, либо переменная, либо кортеж из n термов, перед которым стоит функтор;

(ii) *предикат* — это кортеж из n термов, перед которым стоит предикатный символ.

Формулы могут содержать теперь кванторы:

(iii) *формулой* является либо предикат, либо одно из следующих выражений:

$$\begin{array}{cccc} (\mathbf{F1}) & \mathbf{F1}, \mathbf{F2} & \mathbf{F1} \leftarrow \mathbf{F2} \\ \neg \mathbf{F1} & \mathbf{F1} \vee \mathbf{F2} & \mathbf{F1} \leftrightarrow \mathbf{F2} & Q\mathbf{F1} \end{array}$$

где **F1** и **F2** — произвольные формулы, а **Q** — любой квантор.

И наконец, предложение определяется так:

(iv) *предложение* — это формула, в которой каждое вхождение переменной (если, конечно, они имеются) находится в области действия квантора по этой переменной. Ради краткости принято опускать самые внешние кванторы всеобщности, например

$$\text{не-пусто}(y) \leftrightarrow (\exists x) \text{ принадлежит}(x,y)$$

Переменная y здесь, очевидно, неквантифицирована, и, следовательно, предполагается, что она находится в области действия самого внешнего (опущенного) квантора всеобщности $(\forall y)$. Таким образом, это имеющееся в виду предложение согласуется с правилом (iv). Термы, предикаты, формулы и предложения, не содержащие переменных, называются *основными*.

Множество всех предложений, построенных согласно правилам (i)–(iv), образует *язык логики первого порядка*. В этом языке термы предоставляют средства для обозначения интерес-

сующих нас индивидуумов; при этом используются либо конкретные имена вида *СЛОЖИТЬ(1,2)*, либо обобщенные имена наподобие *СЛОЖИТЬ(x, УДВОИТЬ(СЛОЖИТЬ(2, УДВОИТЬ(y))))*. Предикаты выражают отношения между индивидуумами, которые обозначены с помощью термов, а предложения описывают логические свойства этих отношений. В этом языке могут быть сформулированы все вычислительные задачи (т. е. задачи, решаемые компьютером). В каждой конкретной реализации языка логического программирования существуют соглашения, точно описывающие виды предложений, которые могут входить в программы, а также имеющиеся словари для построения имен.

I. 6. Примеры представлений

В следующих примерах демонстрируется использование предложений для описания некоторых известных структур данных и их свойств.

Пример 1. Описать список $L = (A, B, C, D)$

(a) Будем использовать точку в качестве двухместного функциона для построения термов вида $.(u, y)$. Каждый такой терм может служить именем списка, в котором первым элементом является u , а y обозначает оставшуюся часть списка. Чтобы упростить обозначения, вместо префиксной записи $.(u, y)$ мы будем пользоваться бескобочной инфиксной записью $u.y$. Константа *NIL* может быть именем пустого списка, а предикат список(z, x) может означать, что z есть имя списка x . В этом случае одного предложения

список($L, A.B.C.D.NIL$)

достаточно для того, чтобы описать L как список (A, B, C, D) . По существу в этом предложении одному и тому же списку сопоставляются два имени: L и $A.B.C.D.NIL$, причем второе является более информативным, поскольку показывает, что список состоит из индивидуумов с именами A, B, C, D и *NIL*.

(b) В другом способе описания списка используется предикат $\in(u, i, x)$, означающий, что элемент u занимает позицию с номером i в списке x . Следующие четыре предложения:

- $(A, 1, L)$
- $(B, 2, L)$
- $(C, 3, L)$
- $(D, 4, L)$

все вместе дают другое описание списка L . Они напоминают традиционный программистский метод (хотя и не вполне аналогичны ему) присваивания элементов линейному массиву с помощью операторов вида

$$\begin{aligned}L(1) &:= 'A' \\L(2) &:= 'B' \\L(3) &:= 'C' \\L(4) &:= 'D'\end{aligned}$$

(с) Еще один способ состоит в использовании предиката $\text{след}(u, v, x)$, означающего, что элементы u и v являются соседними в списке x , причем v следует за u . Три предложения

$$\begin{aligned}\text{след } (A, B, L) \\ \text{след } (B, C, L) \\ \text{след } (C, D, L)\end{aligned}$$

также описывают L .

Пример 2. Описать матрицу

$$M = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

(а) Пусть предикат $\text{элем}(u, i, j, x)$ означает, что элемент u стоит в строке i и столбце j матрицы x . Тогда M можно описать четырьмя предложениями

$$\begin{aligned}\text{элем}(1, 1, 1, M) \\ \text{элем}(2, 1, 2, M) \\ \text{элем}(3, 2, 1, M) \\ \text{элем}(4, 2, 2, M)\end{aligned}$$

(б) Другая возможность — использовать предикат $\text{строка}(i, z, x)$, означающий, что i -я строка матрицы x является списком элементов z . В этом случае для описания матрицы достаточно двух предложений:

$$\begin{aligned}\text{строка}(1, 1..2.NIL, M) \\ \text{строка}(2, 3..4.NIL, M)\end{aligned}$$

(с) Аналогично можно описать M с помощью предиката $\text{столбец}(j, z, x)$:

$$\begin{aligned}\text{столбец}(1, 1..3.NIL, M) \\ \text{столбец}(2, 2..4.NIL, M)\end{aligned}$$

Пример 3. Описать отношение принадлежности \in элемента u списку x , используя при этом каждое из трех представлений списков, полученных в примере 1; предикат $\in(u, x)$, означающий, что элемент u является членом списка x .

чающий, что u есть элемент x , можно записывать в более простой инфиксной форме $u \in x$

$$(a) u \in x \leftrightarrow (\exists v \exists y) (\text{список}(x, v.y), (uv = \vee u \in y)) \quad \blacksquare$$

Здесь $=$ есть имя отношения тождества, которое состоит из всех пар одинаковых индивидуумов.

$$(b) u \in x \leftrightarrow (\exists i) \in(u, i, x)$$

$$(c) u \in x \leftrightarrow (\exists v) \text{след}(u, v, x) \vee (\exists v) \text{след}(v, u, x) \vee \\ \vee \text{список}(x, u.NIL)$$

Заметим, что в методе (c) нельзя обойтись применением одного только отношения след, поскольку с его помощью нельзя описывать единичные списки (имеющие ровно один элемент) — поэтому приходится пользоваться также отношением **список**.

Пример 4. Для каждого из трех полученных выше представлений матриц описать отношение между исходной матрицей и транспонированной; при этом можно использовать функтор T , дающий имя $T(x)$ матрице, транспонированной по отношению к матрице x .

$$(a) \text{элем}(u, i, j, T(x)) \leftrightarrow \text{элем}(u, j, i, x)$$

$$(b) \text{строка}(i, z, T(x)) \leftrightarrow \text{столбец}(i, z, x)$$

$$(c) \text{столбец}(i, z, T(x)) \leftrightarrow \text{строка}(i, z, x)$$

В связи с этими примерами возникает множество вопросов. В каком смысле написанные предложения представляют имеющиеся в виду понятия? Являются ли эти представления корректными и полными? Какую роль они играют в решении вычислительных задач, и почему одно множество предложений может быть предпочтительнее другого? Мы займемся этими вопросами позднее, когда будем обсуждать интерпретацию предложений, преобразование их с помощью логического вывода, а также их значение для вычислений.

Прежде чем перейти к этим вопросам, читателю, впервые сталкивающемуся с математической логикой, следует сначала поупражняться в чтении логических предложений, соотнося их неформально с утверждениями из более знакомых символьческих систем, таких как математика и естественный язык. Логика оказывается довольно простой и единообразной формой записи для выражения таких понятий, как в примере 3 (c), где с ее помощью формализуется интуитивное представление о том, что элемент принадлежит списку тогда и только тогда, когда он либо является предшественником, либо последователем некоторого другого элемента, либо вообще является единственным элементом в списке.

I.7. Интерпретация предложений

После того как знания представлены в какой-либо формальной системе, подобной логике, обычно начинают посредством некоторого строго определенного процесса рассуждений изучать логические следствия, вытекающие из этих знаний. Фундаментальным понятием, лежащим в основе таких рассуждений, является понятие *логического следствия*. Мы говорим о некотором заключении, логически следующем из некоторого множества допущений. На содержательном уровне это означает, что если допущения «истинны» в соответствии с какой-либо их интерпретацией, то заключение также должно быть истинным. Вопреки распространенному мнению интуитивные понятия истинности и ложности не являются внутренне присущими свойствами логики. Тем не менее сопоставление этих понятий с логикой дает удобный способ описания отношений между предложениями. Чтобы ввести точное определение логического следствия, мы объясним здесь, что такое интерпретация.

Для дальнейших рассмотрений будет полезно обратиться к конкретному примеру. Пусть у нас имеются сначала два предложения:

положительное(ЕДИНИЦА)

и

($\forall x$) (положительное(УДВОИТЬ(x)) \leftarrow положительное(x))

Из этих предложений, как оказывается, логически следует третье предложение

положительное(УДВОИТЬ(ЕДИНИЦА))

Мы можем сделать этот факт более очевидным, переводя данные предложения на русский язык и образуя умозаключение

если верно, что

ЕДИНИЦА — положительное число

и для всех x число x , умноженное на два, — положительное, если x — положительное.

то верно, что

ЕДИНИЦА, умноженная на два, — положительное число

Несомненная очевидность полученного умозаключения объясняется нашей естественной способностью к неформальным рассуждениям на повседневном языке. Более того, символы, встречающиеся в предложениях, автоматически вызывают в памяти знакомые представления о числах и об их арифметических свойствах. Мы легко можем убедиться в корректности этого умозаключения, связывая первые два предложения с законами арифметики, которые мы уже считаем истинными, и рас-

суждая затем в соответствии со здравым смыслом, чтобы сделать вывод об истинности третьего предложения.

Подобного поверхности рассмотрения обычно бывает достаточно для того, чтобы решить, следуют или нет некоторые заключения из заданных посылок. Очевидно, однако, что на нем сказываются неточности в интерпретации естественного языка. Что, к примеру, точно означают слова «если», «и», «для всех» и «истинно»? Зависит ли корректность этого умозаключения от того, что предложения интерпретируются как законы арифметики? Иные примеры могли бы быть не столь очевидными, и мы вынуждены были бы рассматривать эти вопросы более тщательно.

Формальное определение логического следствия на понятийном уровне очень простое, хотя технически довольно трудоемкое. На этом этапе читатель может без существенного ущерба для себя перейти прямо к разд. I. 9, в котором рассматривается логический вывод, и полагаться на свое содержательное понимание логического следствия. Оставшуюся часть данного раздела, а также следующий раздел при необходимости можно прочитать позднее.

Основная задача при формальном определении логического следствия — установить точное понятие интерпретации. Для того чтобы интерпретировать множество предложений S , мы выберем сначала *область интерпретации*, представляющую собой произвольное множество индивидуумов D . Мотивировка здесь такова: индивидуумы будут сопоставляться именам, входящим в S . Далее, каждому функтору из S ставится в соответствие некоторая функция, определенная на D , а каждому предикатному символу из S — некоторое отношение на D . В результате этого предложения из S могут интерпретироваться как высказывания об области D . Сказать теперь, что некоторое предложение из S истинно (или ложно) относительно выбранной интерпретации, — это значит сказать, что истинно (или ложно) соответствующее высказывание о D . Таким образом, посредством установления связи с внешним миром индивидуумов предложения приобретают истинностно-функциональные значения. Все эти понятия можно сделать очень точными после следующих предварительных определений:

(i) пусть D^n для каждого $n \geq 1$ обозначает множество всех кортежей из n индивидуумов, принадлежащих D ;

(ii) *отображением* из одного множества в другое называется всякое множество пар (x, y) , такое что x принадлежит первому множеству, y — второму, и каждый элемент x встречается в качестве первого аргумента пары ровно один раз; мы говорим тогда, что x отображается в y ;

(iii) истинностное значение, или просто значение — это любой из двух символов t и f , которые читаются как «истина» и «ложь» соответственно.

Интерпретация множества предложений S над областью D состоит тогда из трех *соответствий* $A1$, $A2$ и $A3$, где

$A1$ сопоставляет каждой константе из S некоторый индивидуум из D ;

$A2$ сопоставляет каждому n -местному функтору из S некоторое отображение из D^n в D ;

$A3$ сопоставляет каждому n -местному предикатному символу из S некоторое отображение из D^n в множество $\{t, f\}$.

Пусть, к примеру, S состоит из уже встречавшихся раньше предложений

положительное(*ЕДИНИЦА*)

($\forall x$) положительное(*УДВОИТЬ*(x)) \leftarrow положительное(x)

Произвольным образом выберем теперь область интерпретации и соответствия $A1$, $A2$ и $A3$. Пусть, например,

D — это множество натуральных чисел $1, 2, 3 \dots$ и т. д.;

$A1$ сопоставляет константе *ЕДИНИЦА* число 1 ;

$A2$ сопоставляет функтору *УДВОИТЬ* отображение $\{(1, 2), (2, 4), (3, 6), \dots\}$ и т. д.;

$A3$ сопоставляет предикатному символу *положительный* отображение $\{(1, t), (2, t), (3, t), \dots\}$ и т. д.}.

Интерпретация множества S , таким образом, полностью определена, хотя, по-видимому, мы достигли еще немногого. Интерпретация не имеет никакого значения до тех пор, пока мы не используем ее для оценки каждого предложения из S с помощью истинностных значений t и f . Этого можно достичь, оценивая формулы, из которых построены предложения, в соответствии со следующими правилами (здесь F — произвольная формула, входящая в рассматриваемое предложение).

(a) Если F — предикат без переменных, то сначала заменяем каждую константу из F на индивидуум, который сопоставляется ей, согласно $A1$, затем заменяем каждый структурированный терм тем индивидуумом, в который, согласно $A2$, отображается кортеж аргументов этого терма; в результате произведенных замен получается предикат вида $p(d)$, где d — кортеж из индивидуумов; значение F определяется тогда как истинностное значение, в которое d отображается согласно $A3$.

(b) Если F имеет вид $(\forall x)F'$, где F' — формула, содержащая x , то значением F является t в том случае, когда t — значение каждого примера формулы F' , получаемого одновремен-

ной заменой всех вхождений x некоторым индивидуумом из D ; в противном случае значением F является f .

(с) Если F имеет вид $(\exists x)F'$, где F' — формула, содержащая x , то значением F является t в том случае, когда по крайней мере один пример формулы F' , получаемый одновременной заменой всех вхождений x некоторым индивидуумом из D , принимает значение t ; в противном случае значением F является f .

(д) Если F имеет вид $\neg F_1$ или F_1, F_2 , или $F_1 \vee F_2$, или $F_1 \leftarrow F_2$, или $F_1 \leftrightarrow F_2$, где F_1 и F_2 — формулы, то значение формулы F определяется по приводимой ниже таблице истинности в соответствии со значениями F_1 и F_2 . Заметим, что в

F_1	F_2	$\neg F_1$	F_1, F_2	$F_1 \vee F_2$	$F_1 \leftarrow F_2$	$F_1 \leftrightarrow F_2$
t	t	f	t	t	t	t
t	f	f	f	t	t	f
f	t	t	f	t	f	f
f	f	t	f	f	t	t

правилах (б) и (с) точный смысл понятий «для всех» и «существует» определяется через область интерпретации D , в то время как в правиле (д) связки определяются таким образом, чтобы их смысл соответствовал смыслу содержательных аналогов этих связок в естественном языке. Мы поясним применение этих правил, рассмотрев наш пример вместе с уже предложенной выше интерпретацией. Первое предложение

положительное(ЕДИНИЦА)

является формулой вида (а). С помощью соответствия А1 мы заменяем имя **ЕДИНИЦА** индивидуумом I и получаем **положительное** (I). Согласно А3 кортеж I отображается в истинностное значение t , которое, следовательно, и является значением первого предложения.

Второе предложение — это формула вида (б), где F' есть формула

положительное(УДВОИТЬ(x)) \leftarrow положительное(x)

Требуется оценить примеры F' при всех возможных заменах переменной x индивидуумами из D . Рассмотрим только один пример, получаемый заменой x на I :

положительное(УДВОИТЬ(I)) \leftarrow положительное(I)

Согласно правилу (а), значение **положительное** (I) есть t . Значение **положительное** (**УДВОИТЬ** (I)) совпадает со значением

предиката **положительное** (2), поскольку для функтора **УДВОИТЬ** кортеж 1, согласно А2, отображается в число 2, и, стало быть, структурированный терм **УДВОИТЬ** (1) заменяется на 2. Согласно правилу (а), значение предиката **положительное** (2) также *t*. Таким образом, рассматриваемый пример формулы F' имеет вид $F1 \leftarrow F2$, причем F1 и F2 принимают значение *t*. Применяя правило (д), получаем иаконец, что истинностное значение нашего примера есть *t*. Более того, все другие примеры также принимают значение *t* независимо от того, какой индивидуум из D подставляется вместо *x*, и потому, согласно правилу (б), истинностное значение второго предложения также есть *t*.

Таким образом, в рассматриваемом примере каждое предложение из S принимает в данной интерпретации значение *t*. Можно, однако, так подобрать область D и соответствия A1, A2 и A3, чтобы истинностные значения были другими. Как мы уже подчеркивали в начале этого раздела, предложения сами по себе не являются «истинными» или «ложными». Значения *t* и *f* не более чем символы, сопоставляемые предложениям посредством этого конкретного метода интерпретации. Нельзя, однако, сказать, что эти значения не имеют никакого смысла; их роль станет ясной из следующего раздела.

I.8. Логическое следствие

Теперь мы можем точно определить понятие логического следствия. Мы будем говорить, что предложение выполняется в интерпретации над некоторой областью D тогда и только тогда, когда это предложение принимает в данной интерпретации значение *t*. Множество предложений S выполняется в интерпретации тогда и только тогда, когда в ней выполняется каждое предложение из S. Наконец, мы говорим, что S логически влечет (или просто влечет) некоторое предложение s тогда и только тогда, когда для каждой интерпретации над произвольной областью из выполнимости в ней множества предложений S следует выполнимость в ней предложения s¹⁾.

Вернемся снова к нашему примеру. Как было уже показано, множество S выполняется в выбранной интерпретации над множеством натуральных чисел. Легко установить (читатель сделает это в качестве упражнения), что третье предложение

s: положительное(УДВОИТЬ(ЕДИНИЦА))

¹⁾ В этом случае говорят также, что из S логически следует предложение s или что s является логическим следствием множества предложений S. — Прим. перев.

также выполняется в данной интерпретации. Более того, оказывается, что в каждой интерпретации, в которой выполняется множество предложений S , выполняется также и предложение s независимо от выбираемой области интерпретации. Мы говорим поэтому, что множество предложений

положительное (ЕДИНИЦА)
 $(\forall x)(\text{положительное}(\text{УДВОИТЬ}(x)) \leftarrow \text{положительное}(x))$

логически влечет предложение

положительное(УДВОИТЬ(ЕДИНИЦА))

Это отношение между S и s мы будем записывать в виде $S \models s$, где символ \models читается как «логически влечет».

Смысл отношения \models заключается в том, что если $S \models s$ имеет место, то это происходит только благодаря внутреннему устройству самих предложений независимо от того, что могут обозначать составляющие их части в соответствии с той или иной интерпретацией. Это отношение, таким образом, не зависит от связей, которые могли бы быть установлены между языком и миром индивидуумов. Использование определенных выше интерпретаций, приписывающих значения, — это лишь один способ образования понятия логического следствия, которое, в конечном счете, от этих интерпретаций не зависит.

Связь логического программирования с понятием логического следствия определяется тем способом, которым задачи формулируются и решаются. Рассмотрим типичную задачу решения некоторых уравнений. Средствами логического программирования можно было бы составить множество предложений S , описывающее эти уравнения и критерии их решения. Компьютер выведет затем из S новое предложение s , являющееся логическим следствием S и определяющее решение данных уравнений. Таким образом, логический базис, лежащий в основе решения задач, становится здесь более явным, чем при использовании традиционных языков программирования.

К счастью, на практике для установления отношения $S \models s$ не требуется исследовать различные области и интерпретации, поскольку существуют гораздо более простые методы, основанные на логическом выводе. Поэтому не столь важно, чтобы читатель полностью овладел всеми дававшимися до сих пор формальными определениями. Фактически почти все аспекты формальной системы логического программирования можно было бы представить, даже не упоминая о логическом следствии, хотя это, возможно, не дало бы полного понимания сущности предмета.

I.9. Логический вывод

Логический вывод — это процесс получения некоторого предложения s , исходя из множества предложений S , путем применения одного или нескольких *правил вывода*. Цель вывода состоит, как правило, в том, чтобы показать справедливость отношения $S \models s$ (т. е. что S логически влечет s). В данном разделе вводится очень простой метод построения вывода, на котором основывается обычный способ обработки логических программ. В этом методе используется только одно правило вывода, называемое *результатом*. Каждое применение правила называется *шагом вывода*. Для наших целей будет достаточно применять правило резолюции только к предложениям трех типов, которые называются *отрицаниями*, *фактами* и *импликациями*. Они устроены следующим образом:

отрицание: $\neg(A_1, \dots, A_n)$

факт: A

импликация: $A \leftarrow B_1, \dots, B_m$

(Здесь $A, A_1, \dots, A_n, B_1, \dots, B_m$ — произвольные предикаты.)

В импликации предикат, стоящий слева от связки \leftarrow , называется *консеквентом*, а предикаты, стоящие справа от \leftarrow , называются *антecedентами*. Факты можно рассматривать как импликации, не имеющие антecedентов.

Резолюцию проще всего ввести, рассмотрев сначала одну из наиболее простых ее форм. Допустим, что у нас имеются отрицание и импликация:

отрицание $S_1 : \neg A$

импликация $S_2 : A \leftarrow B$

где предикат A из S_1 совпадает с консеквентом A из S_2 . В результате одного шага резолютивного вывода мы получим из S_1 и S_2 новое предложение

$s : \neg B$

На этом шаге предложения S_1 и S_2 называются *родительскими предложениями*, а s называется *результативной*, которая получается в результате применения резолюции (или резольвирования) к S_1 и S_2 . Резолюция в этом простом случае соответствует стандартному (и очень важному для наших целей) пропозициональному правилу вывода, называемому *modus tollens*, суть которого выражается следующим умозаключением:

допуская, что не A и A если B ,
выводим не B .

Читателю, который ранее не сталкивался с правилом *modus tollens*, следует полностью убедиться в том, что это умозаключение интуитивно оправдано, прежде чем двигаться дальше.

Рассмотрим теперь еще более простой случай, в котором S1 является отрицанием, а S2 — фактом

$$\begin{array}{l} \text{отрицание } S1 : \neg A \\ \text{факт } S2 : A \end{array}$$

Применяя к этим родительским предложением правило резолюции, мы выводим в качестве резольвенты *пустое отрицание*

$$S : \square$$

которое обозначается символом \square и означает *противоречие*. Таким образом, резолюцией здесь является следующее рассуждение:

допуская, что не A и A ,
выводим противоречие

Для того чтобы понять назначение таких выводов в контексте решения задач, рассмотрим простую задачу, связанную с логикой понятий «давать» и «получать». Пусть предикат дает (x, y, z) означает, что « x дает y некоторому объекту z », в то время как другой предикат получает (y, z) означает, что « y получает z ». Допустим далее, что некоторые знания об этих отношениях выражаются двумя предложениями:

$$\begin{array}{l} S2 : \text{получает}(VY, СИЛА) \leftarrow \text{дает}(ЛОГИКА, СИЛА, VY) \\ S3 : \text{дает}(ЛОГИКА, СИЛА, VY) \end{array}$$

Задача, которую нужно решить, состоит в том, чтобы ответить на вопрос:

получаете ли *VY* *СИЛУ*?

Когда используется обычная система логического программирования, такой вопрос представляется в виде отрицания:

$$S1 : \neg \text{получает}(VY, СИЛА)$$

и система должна опровергнуть это отрицание при помощи других предложений; опровергается отрицание путем демонстрации того, что его допущение ведет к противоречию. Этот подход является общепринятым в математике. Его называют *доказательством от противного* или *reductio ad absurdum*.

Представим себе тогда, что логическая программа составлена из трех предложений S1, S2 и S3, и она подается на вход системы резолютивного вывода, реализованной на ЭВМ.

Система применит правило резолюции к S1 и S2 и получит резольвенту

$s : \neg \text{дает}(ЛОГИКА, СИЛА, ВЫ)$

затем применит резолюцию к s и S3 и получит противоречие

$s' : \square$

Для доказательства противоречивости входных предложений S1, S2 и S3 оказалось достаточно, таким образом, двух шагов вывода. Если предположить, что мы не отказываемся от предложений S2 и S3 и сами они не являются противоречивыми, то отсюда вытекает, что они совместно противоречат S1, т. е. подтверждают отрицание предложения S1, а именно предложение

получает(ВЫ, СИЛА)

Следовательно, ответом для исходной задачи является «да».

Резолюция, порождающая последовательность отрицаний, такую как $(S1, s, s')$ в только что рассмотренном примере, называется *резолюцией сверху вниз*; эта терминология объясняется в разд. I. 14.

Реальные программы содержат, как правило, значительно более сложные предложения, чем те, которые мы рассматривали. Отрицания, например, могут иметь несколько предикатов, а импликации — несколько антецедентов. Поэтому более общим является случай, когда родительские предложения имеют вид

$S1 : \neg(A_1, \dots, A_n)$

$S2 : A_k \leftarrow B_1, \dots, B_m$ (где $1 \leq k \leq n$)

Здесь некоторый предикат A_k из отрицания S1 совпадает с коинсеквентом из импликации S2. В этой ситуации шаг вывода заменяет A_k в S1 на антецеденты из S2, и в качестве резольвенты получается отрицание

$s : \neg(A_1, \dots, A_{k-1}, B_1, \dots, B_m, A_{k+1}, \dots, A_n)$

Рассмотрение следующего простого содержательного примера, в котором $n=3$ и $m=1$, поможет лучше понять данное правило:

допуская, что не (темно и зима и холодно)

и что зима если январь

выводим, что не (температура низкая и январь и темно)

В том случае, когда отрицание S1 имеет такой же вид, как и выше, а S2 — это просто факт

$S2 : A_k$

причем A_k является одним из предикатов в S1, шаг вывода только вычеркивает A_k из S1, и в результате получается ре-

зольвента

$$s : \neg(A_1, \dots, A_{k-1}, A_{k+1}, \dots, A_n)$$

Снова приведем содержательный пример:

допуская, что не (темно и зима и холодно)
и что зима
выводим, что не (температура низкая и темно и холодно)

Во всех рассмотренных до сих пор случаях предварительным условием возможности выполнения шага вывода является совпадение некоторого предиката из отрицания S1 с предикатом-консеквентом из импликации или факта S2. Однако, как демонстрируется в следующем разделе, резолюция оказывается значительно более общим правилом вывода.

I. 10. Общая резолюция сверху вниз

Рассмотрим два следующих родительских предложения:

$$\begin{aligned} S1 &: \neg \text{получает}(BYI, y) \\ S2 &: \quad \text{получает}(x, СИЛА) \leftarrow \text{дает}(z, СИЛА, x) \end{aligned}$$

Они содержат три переменные x , y и z , которые (неявно) универсально квантифицированы. Так, например, предложение S1 утверждает, что

для всех y *BYI* не получаете y

Вспомним, что в разд. I. 7 выражение «для всех» понималось как «для всех индивидуумов из какой-либо области, выбранной для интерпретации предложений». В каждой интерпретации предложений S1 и S2 над указанной областью по крайней мере один индивидуум будет связан с именем *СИЛА*, и поэтому непосредственным следствием S1 является более конкретное предложение, в котором говорится именно об этом индивидууме:

$$S1' : \neg \text{получает}(BYI, СИЛА)$$

т. е.

BYI не получаете *СИЛУ*

Аналогичным образом, рассматривая предложение S2 и выбирая для x индивидуум с именем *BYI*, получаем более конкретное предложение:

$$S2' : \text{получает}(BYI, СИЛА) \leftarrow \text{дает}(z, СИЛА, BYI)$$

т. е.

для всех z *BYI* получаете *СИЛУ* если z дает *СИЛУ* *ВАМ*

Теперь у нас имеются два предложения $S1'$ и $S2'$, которые удовлетворяют предварительному условию: в отрицании присутствует предикат, совпадающий с консеквентом импликации. Поэтому резольвентой $S1'$ и $S2'$ будет

$$s : \neg \text{дает}(z, \text{СИЛА}, ВЫ)$$

Предикат **получает** ($ВЫ, СИЛА$) называется *общим примером родительских предикатов*

$$\text{получает}(ВЫ, y)$$

и

$$\text{получает}(x, \text{СИЛА})$$

Это понятие определяется следующим образом.

Подстановкой Θ называется всякое множество присваиваний вида $x := t$, где x — переменная, а t — терм, причем каждой переменной присваивается не более одного значения. Применение подстановки Θ к произвольному выражению E , например к предикату, заключается в замене переменных из E на термы, которые, согласно Θ , присваиваются этим переменным. Каждая переменная из E , не упомянутая в Θ , остается без изменений, а присваивания из Θ переменным, не входящим в E , не выполняются. Результат применения Θ к E обозначается через $E\Theta$ и называется *подстановочным примером* E . Если применение Θ к двум выражениям $E1$ и $E2$ дает одинаковые подстановочные примеры, то выражение $E1\Theta (= E2\Theta)$ называется *общим примером* $E1$ и $E2$, а подстановка Θ называется тогда *унификатором* (или унифицирующей подстановкой для выражений $E1$ и $E2$).

В нашем примере родительские предикаты обладают унификатором

$$\Theta = \{x := ВЫ, y := \text{СИЛА}\}$$

который присваивает переменным x и y в качестве значений темы $ВЫ$ и СИЛА соответственно. На практике обычно унификаторы можно легко определять, сравнивая по очереди соответствующие аргументы предикатов и выписывая те присваивания термов переменным, которые сделали бы эти аргументы одинаковыми. Для поиска унификаторов в сложных случаях и для реализации на ЭВМ имеются систематические алгоритмы унификации (см. гл. VII). На этом этапе будет, вероятно, полезно рассмотреть несколько простых примеров унификации и резолюций, которые приводятся ниже.

- (i) Родительские предикаты — $p(5)$ и $p(5)$; их унификатором Θ является пустое множество (не заменяется ни одна из переменных);
общий пример — $p(5)$.

- (ii) Родительские предикаты — $p(x)$ и $p(5)$;
унификатор — $\Theta = \{x := 5\}$;
общий пример — $p(x)\Theta = p(5)\Theta = p(5)$.
- (iii) Родительские предикаты — $p(x)$ и $p(y)$;
унификатор — $\Theta = \{x := y\}$;
общий пример — $p(y)$.
- (iv) Родительские предикаты — $p(x, x)$ и $p(5, y)$; сравнение первых аргументов дает присваивание $x := 5$, а сравнение вторых аргументов — $y := x$, т. е. с учетом первого присваивания — $y := 5$;
унификатор — $\Theta = \{x := 5, y := 5\}$;
общий пример — $p(5, 5)$.
- (v) Родительские предикаты — $p(f(x), f(5), x)$ и $p(z, f(y), y)$; сравнивая, как и в предыдущем примере, аргументы слева направо и учитывая уже найденные присваивания, получаем унификатор
 $\Theta = \{z := f(5), y := 5, x := 5\}$;
общий пример — $p(f(5), f(5), 5)$.
- (vi) Родительские предложения $S1 : \neg p(5, y)$
 $S2 : p(x, x) \leftarrow q(x)$

унификатор родительских предикатов —

$\Theta = \{x := 5, y := 5\}$;

резольвента — $s : \neg q(5)$

при допущении $S2$ отрицать (посредством $S1$) существование y , такого что справедливо $p(5, y)$, — это значит выбрать $y := 5$ и отрицать (посредством s), что справедливо $q(5)$.

В общем случае унификатор двух предикатов не обязан быть единственным возможным. Мы рассмотрим сейчас пример, когда существуют несколько различных унификаторов. На этот раз родительскими предложениями будут следующие:

$$\begin{aligned} S1 &: \neg \text{поряд}(1.x) \\ S2 &: \text{поряд}(u.v.y) \leftarrow u < v, \text{поряд}(v.y) \end{aligned}$$

Здесь предложение $S2$ выражает свойство упорядоченных списков, таких как $(1, 2, 3)$. Списки представляются структурированными термами, построенными с помощью функтора $.$ и константы NIL (см. разд. I.6). Предикатный символ $<$ является именем отношения порядка. Список вида $u.v.y$ имеет u и v в качестве своих первых двух элементов, а y — это оставшаяся часть списка. В предложении $S2$ утверждается, что такой список будет упорядоченным, если $u < v$ и подсписок $v.y$ является упорядоченным. В предложении $S1$ для любого x отрицается, что список вида $1.x$ упорядочен.

Для того чтобы посмотреть, можно ли применить правило резолюции к родительским предложениям, мы ищем унификатор для предикатов

поряд(I.x) и поряд(u.v.y)

Поскольку предикатные символы у них совпадают, остается проунифицировать аргументы $I.x$ и $u.v.y$. Легко видеть, что подходящим оказывается унификатор

$$\Theta = \{u := 1, x := v.y\}$$

Адекватность подстановки Θ станет, возможно, более прозрачной, если мы рассмотрим ее применение к двум нашим термам, записанным в префиксной форме, а именно к термам (I, x) и $(u, (v, y))$. При таком выборе унификатора Θ из предложений S_1 и S_2 мы получаем резольвенту

$$s : \neg(I < v, \text{поряд}(v,y))$$

в чем читатель может убедиться, применяя Θ к S_1 и S_2 и получая более конкретные предложения S'_1 и S'_2 , как это делалось в предыдущем примере, а затем резольвируя их непосредственно.

Другим возможным унификатором для предложений S_1 и S_2 является

$$\Theta' = \{u := 1, x := 2.NIL, v := 2, y := NIL\}$$

Он дает иную резольвенту:

$$s' : \neg(I < 2, \text{поряд}(2.NIL))$$

Унификатор Θ оказывается более общим, чем Θ' , в том смысле, что Θ осуществляет менее конкретные подстановки термов вместо переменных, чем Θ' . На самом деле Θ является *наиболее общим унификатором*, с помощью которого можно унифицировать два наших предиката *поряд*: он требует только то, чтобы терм, подставляемый вместо x , имел вид $v.y$, где переменные v и y остаются неконкретизированными, в то время как любой другой унификатор накладывал бы подобно Θ' большие ограничения на этот терм, подставляя какие-либо термы вместо переменных v и/или y . Используя резолюцию, мы неизменно употребляем наиболее общий унификатор. Если предикаты унифицируемы, то он всегда существует и притом только один¹⁾.

Детали шага вывода, которого достаточно для выполнения общей резолюции сверху вниз во всех рассматривавшихся до сих пор примерах, можно суммировать теперь следующим образом.

¹⁾ Алгоритм поиска наиболее общего унификатора приводится в разделе VII.4.3. — Прим. перев.

Шаг 1. Сначала во избежание путаницы добьемся того, чтобы отрицание S1 и импликация S2 не содержали общих переменных; с этой целью можно заменить некоторые из переменных на новые, что никоим образом не меняет смысла родительских предложений.

Шаг 2. Выберем в отрицании предикат, у которого предикатный символ совпадает с предикатным символом в консеквенте импликации; если такого предиката нет, то шаг вывода оказывается невозможным.

Шаг 3. Найдем наиболее общий унификатор Θ для выбранного предиката из S1 и консеквента из S2; если унификатора не существует, то шаг вывода невозможен.

Шаг 4. Заменим выбранный предикат в S1 на антецеденты импликации S1 или, если S2 является фактом, просто вычеркнем этот предикат; в результате получается новое (возможно, пустое) отрицание.

Шаг 5. Применим Θ к новому отрицанию (что более экономично, чем сначала применять Θ к родительским предложениям, а затем резольвировать их при помощи общего примера, как это делалось в рассмотренных ранее примерах); в результате получаем резольвенту с предложений S1 и S2.

Значение изложенного в данном разделе материала состоит в том, что все вычислительные задачи можно сформулировать, пользуясь только отрицаниями, импликациями и фактами, а те из задач, которые являются разрешимыми, можно решить с помощью общей резолюции сверху вниз.

I. 11. Решение задач

Теперь возможности резолюции при решении задач будут проиллюстрированы полным примером. Мы рассмотрим задачу, в которой исследуется предшествование элементов в списках, представленных термами. Интересующее нас отношение описывается с помощью предиката *след* (u, v, x), который означает, что элементы u и v являются соседними в списке x , причем u предшествует v .

Для описания отношения *след* достаточно двух предложений. Одно из них — факт — устанавливает, что в каждом списке первый элемент всегда предшествует второму:

$$S1 : \text{след}(u, v, u . v . x)$$

а другое — импликация — описывает предшествование среди остальных элементов списка:

$$S2 : \text{след}(u, v, w . y) \leftarrow \text{след}(u, v, y)$$

Возможно, простейший способ понять адекватность описания отношения след посредством этих предложений — это представить себе ситуацию, когда вы хотите продемонстрировать, что два данных элемента i и u действительно являются последовательными элементами данного списка. Тогда вы либо показали бы, что список имеет вид $i.u.x$, либо показали бы, что он имеет вид $w.y$, а элементы i и u являются последовательными в списке y . Предложения $S1$ и $S2$ соответствуют этим альтернативам, и ими, очевидно, исчерпываются все возможные случаи.

Составление предложений $S1$ и $S2$ — это первый шаг в решении задачи; он заключается в формулировке наших *допущений* о проблемной области. Вообще говоря, допущений может быть намного больше.

На втором шаге нужно выразить конкретную задачу, поставленную на проблемной области. Задачу всегда можно представить как запрос об одном или нескольких отношениях, определенных на этой области. В нашем примере задача — показать, что числа 3 и 4 являются последовательными элементами списка $(1, 2, 3, 4, 5)$. Она эквивалентна запросу, будет ли предикат $\text{след}(3, 4, 1, 2, 3, 4, 5, \text{NIL})$ логически следовать из допущений $S1$ и $S2$.

Обычно в этом случае запрос ставится как исходное отрицание:

$$D1 : \neg \text{след}(3, 4, 1, 2, 3, 4, 5, \text{NIL})$$

Составлением допущений и исходного запроса завершается работа программиста, цель которой — сформулировать задачу. В составленных предложениях выражается все, что он хочет сказать и спросить относительно данной конкретной задачи.

Последний, третий шаг выполняется компьютером, который пытается решить задачу, строя доказательство от противного. Более точно, он строит *вывод сверху вниз*, т. е. начиная с исходного отрицания, порождает последовательность отрицаний ($D1, \dots, Dn$). Если может быть построен вывод, заканчивающийся отрицанием $Dn = \square$, то этот вывод, называемый *успешным выводом*, сразу дает решение поставленной задачи.

Вывод строится путем применения правила резолюции к каждому последующему отрицанию, начиная с $D1$, и какому-либо другому родительскому предложению, выбиравшему из множества допущений $S = \{S1, S2\}$; получаемая в результате резольвента становится очередным отрицанием в выводе. В рассматриваемом примере вывод строится так.

<u>Отрицание</u>	<u>Родительские предложения</u>
D1 : $\neg \text{след}(3,4,1.2.3.4.5.NIL)$	нет
D2 : $\neg \text{след}(3,4, 2.3.4.5.NIL)$	D1, S2
D3 : $\neg \text{след}(3,4, 3.4.5.NIL)$	D2, S2
D4 : \square	D3, S1

Вывод (D1, D2, D3, D4) успешный, так как заканчивается пустым отрицанием \square . Он решает нашу задачу, поскольку в теории резолюции установлен следующий важный результат:

из S логически следует $\neg D1$ тогда и только тогда, когда из S и D1 можно вывести \square

В данном примере предложение $\neg D1$ эквивалентно предикату $\text{след}(3,4,1.2.3.4.5.NIL)$, и, стало быть, вывод \square подтверждает, что это заключение следует из множества допущений S.

Независимо от приведенной выше теоремы о резолюции мы обычно считаем множество S внутренне непротиворечивым (или, что эквивалентно, выполнимым), ибо в противном случае противоречивость S означала бы, что допущения, сделанные о проблемной области, взаимно противоречивы, как бы они ни интерпретировались — и тогда демонстрация при помощи резолюции того факта, что из множества допущений логически следует $\neg D1$, не имела бы непосредственной интерпретации, связанной с решением задачи. Непротиворечивость множества S будет гарантирована всякий раз, когда оно содержит только импликации и/или факты; именно так обстоит дело для всех стандартных логических программ.

Успешный вывод можно трактовать и по-другому, используя тот факт (также установленный в теории резолюции), что каждая резольвента логически следует из своих родительских предложений. Поднимаясь по цепи таких логических следствий, входящих в приведенный выше вывод, легко видеть, что из D1 и S следует \square . Пустое отрицание всегда оценивается как «ложное» в любой интерпретации. Если из D1 и S следует \square , то и в одной интерпретации не могут выполняться одновременно D1 и S, поскольку в противном случае \square должно было бы иметь значение «истина». Отсюда вытекает, что в каждой интерпретации, в которой выполняется S, предложение D1 должно принимать значение «ложь», а значит $\neg D1$ должно принимать значение «истина». Таким образом, установлено, что $S \models \neg D1$.

Не так важно, если эти объяснения покажутся довольно сложными при первом прочтении. Необходимо только усвоить основной результат: построение вывода пустого отрицания при помощи правила резолюции означает, что из множества допу-

щений S логически следует заключение $\neg D1$. Это заключение прямо отвечает на запрос, поставленный исходным отрицанием D1, и потому оно является утверждением о решении задачи. Поскольку исходное отрицание было опровергнуто, успешный вывод называется *опровержением*. Методы построения опровержений называются *процедурами опровержения*.

I.12. Извлечение ответа

В только что рассмотренном примере ответом на поставленную задачу будет просто «да»; относительно данных допущений предикат *след(3,4,1.2.3.4.5.NIL)* является истинным. Однако более часто требуются ответы, сообщающие нам конкретные значения аргументов, при которых некоторые предикаты являются истинными, причем на стадии формулировки задачи эти значения полностью не известны. Оказывается, что такого рода ответы можно извлекать из унификаторов, которые использовались в выводах, порожденных исходной программой.

В качестве примера рассмотрим еще одну задачу о следующем элементе спуска: *найти*, какой элемент z, если, конечно, он имеется, следует непосредственно за числом 3 в списке (1,2,3,4,5). Будет показано, что ответ 4 естественным образом получается в результате применения резолюции. На этот раз подходящим исходным отрицанием является

$$D1 : \neg \text{след}(3, z, 1.2.3.4.5.NIL)$$

Для любого z оно отрицает, что элемент z непосредственно следует за числом 3 в списке 1.2.3.4.NIL. Поскольку переменная z входит в D1, она называется «*проблемной переменной*». Отрицание D1 можно рассматривать как запрос: существует ли какой-либо пример z, при котором данный предикат будет истинным.

Используя те же допущения

$$S1 : \text{след}(u, v, u.v.x)$$

$$S2 : \text{след}(u, v, w.y) \leftarrow \text{след}(u, v, y)$$

что и прежде, мы можем образовать следующее опровержение:

<u>Отрицание</u>	<u>Родительские предложения</u>
D1 : $\neg \text{след}(3, z, 1.2.3.4.5.NIL)$	нет
D2 : $\neg \text{след}(3, z, 1.2.3.4.5.NIL)$	D1, S2
D3 : $\neg \text{след}(3, z, 1.2.3.4.5.NIL)$	D2, S2
D4 : \square	D3, S1

Оно показывает, что из допущений S1 и S2 логически следует $\neg D1$. Записанное полностью предложение D1 имеет вид

$$(\forall z) \neg \text{след}(3, z, 1.2.3.4.5.NIL)$$

Оно логически эквивалентно предложению

$$\neg (\exists z) \text{след}(3, z, 1.2.3.4.5.NIL)$$

(Два предложения называются *логически эквивалентными*, если каждое из них логически влечет другое.) Стало быть, из наших допущений логически следует отрицание последнего предложения, т. е. предложение

$$(\exists z) \text{след}(3, z, 1.2.3.4.5.NIL)$$

которое устанавливает существование некоторого элемента z , следующего непосредственно за числом 3 в списке (1,2,3,4,5). Для того чтобы найти, какой это элемент, необходимо внимательно изучить *протокол связываний* данного вывода, т. е. множество всех тех присваиваний, порожденных унификацией, которые оказывают влияние на значения, получаемые в конце концов проблемными переменными. (Если переменной в качестве значения присваивается некоторый терм, то мы говорим, что он *связывает* эту переменную, или, эквивалентно, что переменная становится связанной с этим термом.) Участвующие в построении выводе (наиболее общие) унификаторы приводятся ниже рядом с образованными с их помощью отрицаниями.

<u>Отрицание</u>	<u>Унификатор</u>
D2	$\{u := 3, v := z, w := 1, y := 2.3.4.5.NIL\}$
D3	$\{u := 3, v := z, w := 2, y := 3.4.5.NIL\}$
D4	$\{u := 3, v := z, z := 4, x := 5.NIL\}$

Проблемная переменная z из D1 на протяжении первых двух шагов вывода остается несвязанной, поскольку первые два унификатора значения ей не присваивают. Поэтому и после первого, и после второго шага вывода протокол связываний остается пустым. Однако на третьем шаге используется унификатор, который связывает переменную z с термом 4, так что протокол связываний является тогда множеством $\{z := 4\}$. Это множество к тому же оказывается *заключительным* состоянием протокола связываний, когда вывод завершается. Таким образом, полученным опровержением и заключительным состоянием проблемной переменной устанавливается, что из допущений S1 и S2 логически следует ответ

$\text{след}(3, z, 1.2.3.4.5.NIL)$, где $z := 4$,

т. е.

$$\text{след}(3, 4, 1.2.3.4.5.NIL)$$

Процесс восстановления значений проблемных переменных по протоколу связываний в выводе называется *извлечением ответа*. В реализации языка логического программирования извлечение ответа производится компьютером автоматически.

Выполнение унификации и извлечение ответа вручную требуют некоторой осторожности. В частности, процесс переименования переменных в родительских предложениях (цель которого — гарантировать, чтобы во время применения резолюции родительские предложения не имели общих переменных) нужно выполнять следующим образом.

Перед каждым шагом

если переименование необходимо,

то переименовывать следует переменные только в родительском допущении, причем так, чтобы новые переменные отличались от тех, которые входят в родительское отрицание, а также от всех тех переменных, которые упоминаются в текущем состоянии протокола связываний.

Данное правило предохраняет от возможной путаницы, происходящей из-за совпадения переменных.

Путаница может возникнуть также в том случае, когда оба унифицируемых предиката содержат переменные, занимающие одну и ту же позицию среди аргументов. Именно это случилось на первом шаге вывода в рассматриваемом примере, когда унифицировались предикаты

след(3,z,1.2.3.4.5.NIL)

и

след(*u*,*v*,*w*.*y*)

Вторым аргументом в каждом из предикатов является переменная. Возникает вопрос: следует ли переменной *v* присваивать значение *z* или, наоборот, переменной *z* присваивать значение *v*? Прежде мы выбирали первый вариант, однако на первом шаге вывода можно было бы использовать также унификатор $\{u := 3, z := v, w := 1, y := 2.3.4.5.NIL\}$, который присваивает *z* значение *v*. По существу это тот же самый унификатор, и в принципе не имеет значения, каким из них пользоваться. На практике выбор второго унификатора приводит к менее компактному протоколу связываний, и, кроме того, возникает необходимость в переименовании некоторых переменных. Ниже приводится вывод и протокол связываний, которые получаются в результате использования второго варианта унификаторов.

<u>Отрицание</u>	<u>Родительские предложения</u>
D1 : $\neg \text{след}(z, 1.2.3.4.5.NIL)$	нет
D2 : $\neg \text{след}(z, v, 2.3.4.5.NIL)$	D1,S2
D3 : $\neg \text{след}(z, v', 3.4.5.NIL)$	D2,S2 (переменная v переименована на v')
D4 : \square	D3,S1 (переменная v переименована на v'')
<u>Отрицание</u>	<u>Текущее состояние протокола связываний</u>
D1	пустое множество (на первом шаге z не связанная)
D2	$\{z := v\}$
D3	$\{z := v, v := v'\}$
D4	$\{z := v, v := v', v' := v'', v'' := 4\}$

Значение, которое в конце концов получает переменная z , как и прежде 4, однако здесь оно выражается через последовательность связываний. Общее правило таково: если требуется принять решение относительно того, какой из переменных следует присвоить значение другой переменной, то наиболее простой результат получается, когда в качестве присваиваемого терма выбирается переменная из отрицания. Так, в данном примере предпочтительнее выбрать присваивание $v := z$, а не $z := v$. Указанное правило имеет также гораздо более глубокое обоснование, исходя из соображений экономии памяти в период исполнения в реализациях языка логического программирования. Этот вопрос будет обсуждаться в гл. VII.

При достаточной практике выполнения вывода сверху вниз вручную вырабатывается привычка выбирать на каждом шаге такие переименования переменных и такие унификаторы, которые необходимы для поддержания достаточно компактного и понятного протокола связываний. Это искусство сравнимо с умением придавать более простую форму длинным математическим доказательствам путем периодического преобразования имен и приведения в порядок накопленных подстановок. Неумение делать это хорошо может привести только к получению менее элегантных выводов, но не к неверному конечному результату.

I. 13. Резюме

Программирование на языке логики требует от программиста ясного понимания отношений, связанных с той задачей, для которой ищется решение. Более того, он должен быть способен выразить свое понимание, формулируя нужные логические свой-

ства этих отношений на простом языке фактов и импликаций. Сама задача всегда может рассматриваться как запрос о содержимом отношений, который ставится в качестве исходного отрицания, открытого для опровержения.

Как только допущения и запрос сформулированы, они появляются на вход системе построения резолютивного вывода, реализованной на ЭВМ. Система пытается ответить на запрос, строя подходящее опровержение. Если на запрос можно получить ответ на основе знаний, закодированных во входных допущениях, то система обязательно его обнаружит.

Ответственность за построение вывода, за управление протоколом связываний и извлечение ответа может быть либо полностью возложена на систему, либо распределена между системой и программистом в зависимости от особенностей конкретной реализации. Каждое вычислительное решение обладает требуемым свойством — оно логически следует из данных допущений, а само вычисление может быть прямо понято как процесс разумных и систематических рассуждений. Очень немногие из существующих формальных систем программирования обладают этими качествами.

I.14. Исторический очерк

Основания символической логики были заложены (довольно несистематически) Булем, Де Морганом и другими в прошлом столетии. Современной, систематической формуларовкой логики первого порядка мы обязаны прежде всего Фреге. Взаимоотношения между логикой и математикой впервые были всесторонне исследованы в знаменитой работе Рассела и Уайтхеда «Principia Mathematica» 1910, где показана адекватность логики для вывода значительной части математики. Превосходное введение в историю и основания современной логики дается в книгах Де Лонга (1970), Ходжеса (1977) и Робинсона (1979).

В течение длительного времени семантика (или «значение») логики оставалась чрезвычайно запутанным предметом, но в конце концов значительная ясность в него была внесена Тарским; прозрачное изложение современного взгляда на логическое доказательство и логическое следствие дается в его статье (Тарский, 1969). В частности, решающую роль в его подходе играет точное понятие интерпретации, определяемое в разд. I.7. Когда некоторое множество предложений выполняется в интерпретации, ее называют моделью этого множества, а рассмотрение логической семантики на основе моделей известно как теория моделей. Значение теоретико-модельной семантики для логического программирования обсуждается в гл. VIII.

Комбинируя множество правил вывода с некоторой стратегией их применения, получаем алгоритм, называемый *процедурой доказательства*, который, очевидно, можно закодировать в виде программы для ЭВМ. Процесс исполнения такой программы с целью порождения логических выводов из логических предложений, выступающих в качестве входных данных, называется *автоматическим доказательством* теорем. Исследования в области автоматического доказательства теорем в течение трех последних десятилетий, причем очень значительные исследования, отражают давнее стремление к систематизации математических доказательств. Первые запрограммированные процедуры доказательства применялись поэтому в основном для доказательства математических теорем. К их созданию побуждала надежда, что компьютеры докажут существенные теоремы, доказательства которых окажутся слишком длинными или слишком трудными, чтобы их можно было получить немеханическими методами; можно было бы ожидать тогда, что компьютеры ускорят темпы математических открытий.

Помимо потенциального вклада в расширение математического знания автоматическое доказательство теорем считалось также важным для тех аспектов искусственного интеллекта, которые связаны с обработкой знаний при помощи логического вывода. Здесь автоматическое доказательство теорем успешно применялось для выполнения таких заданий, как ответы на вопросы, игры, решение задач в пространстве состояний. Успех в этих областях оказался возможным благодаря выразительной синтаксике логики, достаточной для представления знаний, и способности логического вывода обрабатывать их.

Открытие Робинсоном (1965) правила резолюции после долгих усилий многих исследователей, направленных на разработку систематических и эффективных процедур доказательства в логике первого порядка, представляло собой значительный шаг на пути практического применения автоматического доказательства теорем. Резолюция обладает важными свойствами *корректности* и *полноты*. Резолюция корректна в том смысле, что если с ее помощью из множества допущений S и отрицания D выводится \square , то обязательно справедливо отношение $S \vdash \neg D$. Она является полной в том смысле, что если справедливо $S \vdash \neg D$, то \square непременно выводится из S и D . Полнота и эффективность различных резолютивных систем обсуждались в статье Ковальского и Кюнера (1971).

Несмотря на упомянутые свойства, на резолюцию действуют ограничения, накладываемые на доказуемость сущностью самой природы логики. Фундаментальной проблемой, связанный с формальной логической системой, является «проблема обще-

значимости». Предложение системы называется *общезначимым*, если оно выполняется во всех интерпретациях над произвольными областями; проблема общезначимости заключается в том, чтобы решить, будет или нет какое-нибудь заданное предложение общезначимым. Эта проблема затрагивает и логическое программирование, поскольку задача установления справедливости отношения $S \models \neg D$ эквивалентна задаче установления общезначимости предложения $(\neg D \leftarrow S^*)$, где S^* — конъюнкция (получающаяся в результате соединения связкой «и») всех предложений из S . Черчем и Тьюрингом в 1936 году было доказано (подробное описание этих открытых см. в книге Дэвиса (1958)), что проблема общезначимости для логики первого порядка оказывается только *частично разрешимой*; не существует алгоритма, позволяющего по любым S и D решить, справедливо $S \models \neg D$ или нет; но, к счастью, имеются алгоритмы, которые всегда устанавливают справедливость отношения $S \models \neg D$, если оно действительно имеет место. Эти алгоритмы называются *частичными разрешающими процедурами*. Значение всего этого состоит в том, что если мы по невнимательности или в силу какой-то другой причины составили логическую программу, не имеющую опровержений (т. е. неразрешимую программу), и подали ее на вход частичной разрешающей процедуре, основанной на резолюции и реализованной на ЭВМ, то исполнение этой программы, заключающееся в безуспешных попытках вывести \square , может никогда не закончиться. Такой безрезультатный исход в подобных системах всегда возможен; он является прямым следствием теоремы Черча — Тьюринга.

На практике, однако, незавершаемость иногда возникает даже в том случае, когда исполняются разрешимые программы, поскольку, как объясняется в главе V, в стандартной реализации языка логического программирования употребляется особая стратегия управления, которая не позволяет системе использовать полноту резолюции. Поэтому опровержение, в принципе выводимое, может в зависимости от обстоятельств никогда не быть получено из-за того, что исполнение программы с самого начала попадает в ловушку какого-то другого незаканчивающегося вывода. В логическом программировании, стало быть, существует такой же риск, связанный с незавершаемостью исполнения программ, как и в традиционном программировании, хотя причины этого не совсем одинаковы.

Резолюция является более общим правилом, чем та его версия, которая приводилась в этой главе. На самом деле правило резолюции применяется ко всем предложениям, имеющим вид

$$A_1 \vee \dots \vee A_n \leftarrow B_1, \dots, B_m$$

и называемым *дизъюнктами*¹⁾. Можно показать, что каждое множество предложений в логике первого порядка преобразуется в множество дизъюнктов, имеющее в точности такие же свойства выполнимости. Простой алгоритм преобразования дается в книге Нильсона (1971). В основе теории резолюции лежат ранние исследования Эрбрана (1930) процедур доказательства в логике дизъюнктов. В книге Ченя и Ли (1973) дается ясное введение в теорию и приложения резолюции.

Предложения, являющиеся отрицаниями, фактами или импликациями, образуют подкласс класса всех дизъюнктов и называются хорновскими дизъюнктами. Язык логического программирования называют иногда «логикой хорновских дизъюнктов». Хорновские дизъюнкты пригодны фактически для любых целей, связанных с решением задач. Достаточность этого подкласса с точки зрения теории вычислимости обсуждается в гл. VIII.

Вывод следующих друг за другом отрицаний — это только один способ применения резолюции для решения задач, сформулированных в логике хорновских дизъюнктов; его называют *резолюцией сверху вниз*. В настоящее время это наиболее важный вид резолюции, используемый в логическом программировании. Другой способ, при котором вместо отрицаний последовательно порождаются факты, называется *резолюцией снизу вверх*. Можно предложить и другие способы, различным образом комбинирующие эти два подхода. Термины «снизу вверх» и «сверху вниз» описывают направление, в котором шаги вывода следуют между допущениями и заключениями. Вывод сверху вниз начинают строить с предполагаемого заключения решаемой задачи, представленного в виде исходного отрицания. Его пытаются решить, выводя новое предполагаемое заключение (следующее отрицание), и т. д. Этот способ всегда направлен на решение изначально поставленной задачи. Вывод снизу вверх строится в противоположном направлении: новые факты порождаются посредством применения правила резолюции к старым фактам и импликациям. Этому способу не свойственна направленность к какому-либо конкретному предполагаемому заключению, и поэтому эффективно управлять построением такого

¹⁾ Точнее, дизъюнктом (в оригинале *clause*) называется дизъюнкция литер (отсюда и название — «дизъюнкт»); приведенное выше предложение логически эквивалентно дизъюнкту

$$A_1 \vee \dots \vee A_n \vee \neg B_1 \vee \dots \vee \neg B_m.$$

вывода намного труднее. Его можно рассматривать как «ориентированное на допущения» решение задач в противоположность «целенаправленной» сущности метода построения вывода сверху вниз. Первое всестороннее исследование общего решения задач в логике дизъюнктов было выполнено Ковалевским (1974а), который проиллюстрировал много различных стилей как представления знаний, так и рассуждений. Более полное и современное изложение этих вопросов дано в его книге «Логика для решения задач» (1979а).

II. Логические программы

В предыдущей главе теоретические основы логического программирования объяснялись при помощи понятий и терминологии из теории поиска доказательств теорем в логике первого порядка. Однако операционные возможности логических программ можно описать, исходя из вычислительных понятий, уже известных в традиционных языках программирования. К этим возможностям относятся присваивание данных, ввод и вывод вызовов процедуры, проверка условий и ветвление, итерация и рекурсия. Все они, а также некоторые более экзотические механизмы, прямо и естественно возникают из описанного в гл. I резолютивного процесса, и поэтому, к счастью, не требуется вводить никакой новой специальной терминологии. Цель этой главы, следовательно, состоит в том, чтобы просто выявить вычислительные эффекты применения резолюции к логическим программам и таким образом дать читателю некоторое ощущение практического владения формализмом.

II.1. Программы, вычисления и исполнение программ

Мы начнем с того, что заново определим общую структуру логических программ и примем некоторые соглашения относительно обозначений и терминологии. Мы введем также точные понятия вычисления и исполнения, и тем самым получим все необходимое для объяснения поведения программ в ходе вычислений.

II.1.1. Утверждения в программах

В настоящее время используются разнообразные системы обозначений для написания логических программ. Некоторые из них отражают соглашения, принятые в литературе по резолютивному доказательству теорем и существовавшие до возникновения логического программирования. Более современные разрабатываются для удобства создания и использования интерпретаторов логических программ или в интересах програм-

мистов, или в идеале для того и другого. Мы кратко рассмотрим некоторые из них, прежде чем фиксировать те конкретные обозначения, которые и будут использоваться в оставшейся части книги.

Запись дизъюнктов при помощи положительных и отрицательных литер, считающаяся ныне довольно устаревшей, связана с тем, что общий дизъюнкт вида

$$P \vee Q \leftarrow R, S, T$$

логически эквивалентен формуле

$$P \vee Q \vee \neg R \vee \neg S \vee \neg T$$

Предикаты и их отрицания называются соответственно *положительными* и *отрицательными литерами*, и поэтому предыдущий дизъюнкт можно по-другому записать в виде

$$+ P + Q - R - S - T$$

Используя такую систему записи, логическую программу

$$\begin{aligned} & \neg(A, B) \\ & A \leftarrow B, C \\ & B \\ & C \end{aligned}$$

можно было бы представить следующим образом:

$$\begin{aligned} & -A - B \\ & + A - B - C \\ & + B \\ & + C \end{aligned}$$

Это представление обладает определенной алгебраической лаконичностью, но в нем теряется интуитивный смысл отрицаний, фактов и (особенностей) импликаций.

В более современной *стрелочной записи* вместо символа \neg в отрицании используется символ \leftarrow , который присоединяется также справа к каждому факту. Приведенную выше программу можно было бы записать тогда следующим образом

$$\begin{aligned} & \leftarrow A, B \\ & A \leftarrow B, C \\ & B \leftarrow \\ & C \leftarrow \end{aligned}$$

В этом случае можно считать, что все предложения логической программы имеют вид

$$P \leftarrow Q, R, \dots \text{ и т. д.}$$

где предикаты по обе стороны от стрелки могут как присутствовать, так и отсутствовать. Хотя этот очень единобразный способ записи оказался довольно популярным, он представляется слишком аскетическим.

Для наших целей в дальнейшем будет достаточно менее строгих обозначений. Мы будем писать символ \exists в отрицаниях вместо \neg , подчеркивая тем самым, что отрицание — это по существу запрос. В импликациях вместо \leftarrow будет писаться связка **если**¹⁾. Эти обозначения довольно близки к тем, которые используются в некоторых существующих реализациях. Рассматриваемая программа примет тогда следующий вид

$$\begin{array}{c} \exists A, B \\ A \text{ если } B, C \\ B \\ C \end{array}$$

В некоторых системах для обозначения связки «и» употребляется также символ $\&$, но это приводит к излишней громоздкости длинных предложений; здесь мы по-прежнему будем пользоваться запятой.

В дальнейшем все предложения, составляющие программу, будут называться *утверждениями*. Отрицание будет называться *целевым утверждением*²⁾. Поэтому, когда в программе содержится целевое утверждение вида

$$\exists A(x), B(y)$$

мы говорим, что цель (или назначение) программы — ответить на запрос: «При каких примерах (значениях) переменных x и y имеет место $A(x)$ и $B(y)$?»

Импликации и факты будут в дальнейшем называться *процедурами*³⁾. Каждая из них называется процедурой «для» того предикатного символа, с которого она начинается. Например, утверждение

$$A(x) \text{ если } B(x), C(x)$$

является процедурой для **A**.

¹⁾ В переведении на русский язык книге Клоксина и Меллиша (1981) вместо «если» используется символ: \rightarrow . — *Прим. перев.*

²⁾ Следует указать на отличие этой терминологии от принятой в упомянутой книге Клоксина и Меллиша (1981), где целью (goal) или целевым утверждением назывался лишь один атом из отрицания (ниже он будет интерпретироваться как вызов процедуры); все отрицание было названо при переводе целевым дизъюнктом (goal statement). — *Прим. перев.*

³⁾ Импликации часто называют также правилами (см., например, Клоксин, Меллиш (1981)). — *Прим. перев.*

II.1.2. Назначение и структура программ

Структура стандартной логической программы такова: программа — целевое утверждение и произвольное количество процедур

Ни на число, ни на длину утверждений никаких ограничений не накладывается.

Порядок, в котором процедуры появляются в программе, логического значения не имеет, поскольку каждая из них устанавливает некоторый факт о решаемой задаче независимо от остальных. Обычно, однако, принято собирать процедуры для одного и того же предикатного символа в группу, которая называется тогда *множеством процедур* для этого символа. Например,

$$\begin{aligned} & A(x) \text{ если } B(x), C(x) \\ & A(x) \text{ если } C(y), D(x,y) \\ & B(x) \text{ если } D(x,x) \\ & B(x) \text{ если } C(x) \\ & \vdots \\ & \vdots \\ & \text{и т. д.} \end{aligned}$$

Здесь имеются по крайней мере два множества процедур: одно для предикатного символа **A**, а другое — для **B**. (Иногда предпочитают применять слово «процедура» к тому, что мы назвали множеством процедур; отдельные предложения в нем именуются тогда дизъюнктами. Раньше множества процедур иногда назывались «разделами».) Из последующего будет видно, что порядок, в котором процедуры расположены внутри соответствующего им множества процедур, имеет на самом деле значение только в связи с эффективностью вычислений. Точно так же и упорядочение предикатов в целевых утверждениях и процедурах оказывает влияние только на эффективность, но никакого логического значения не имеет.

Для того чтобы обсуждать примеры программ, удобно иногда снабжать их утверждения метками, облегчающими ссылки на них в тексте. Эти метки выбираются произвольным образом и отделяются от утверждений двоеточием. Сами они не являются составными частями утверждений, т. е. не принадлежат языку логического программирования.

Ниже приводится программа нахождения следующего элемента, рассматривавшаяся в предыдущей главе и записанная теперь в наших новых обозначениях.

целевое утверждение G1: ? след(3, z, 1.2.3.4.5.NIL)

процедуры

C1: след(u, v, u.v.x)

C2: след(u, v, w.y) если след(u, v, y)

В этой программе имеется одно множество процедур для предикатного символа *след*, две процедуры в котором снабжены метками C1 и C2. Исходное целевое утверждение помечено с помощью G1. Проблемную переменную *z* в G1 мы будем называть теперь *целевой переменной*.

Данная программа предназначена для того, чтобы решить целевое утверждение G1, используя при этом только процедуры C1 и C2. В терминах, ориентированных на решение задач, решение целевого утверждения заключается в вычислении некоторого значения *t* (которое будет каким-либо термом) целевой переменной *z*, такого что *t* является элементом, следующим непосредственно за числом 3 в списке (1, 2, 3, 4, 5). В терминах отношений решение целевого утверждения G1 заключается в нахождении примера *t* переменной *z*, при котором тройка (3, *t*, 1.2.3.4.5.NIL) принадлежит трехместному отношению с именем *след*. Наконец, в логических терминах оно заключается в нахождении примера *t* переменной *z*, такого что предложение

$$\text{след}(3, t, 1.2.3.4.5.NIL)$$

логически следует из процедур C1 и C2.

II.1.3. Исполнение программ

Логическая программа начинает исполняться после того, как она подается на вход *логическому интерпретатору*. Интерпретатор представляет собой программу, способную строить резолютивные выводы, как правило, методом «сверху вниз». Получив входную логическую программу, он предпринимает обычные шаги, необходимые для исполнения программы в режиме интерпретации: так, он осуществляет синтаксический контроль входных утверждений; хранит их в центральной памяти в соответствующей упрощенной, компактной и доступной форме; переводит входное целевое утверждение во внутреннее представление и затем начинает процесс построения вывода путем последовательного применения правила резолюции к текущему целевому утверждению и некоторой родительской процедуре, выбираемой из хранящейся в памяти версии входной программы. Если интерпретатору удается вывести пустое отрицание \square , означающее, что решение получено, то он выдает какое-либо сообщение об этом вместе с найденными значениями целевых переменных. Распечатка, полученная в результате исполнения программы о следующем элементе, выглядела бы примерно так:

РЕШЕНИЕ НАЙДЕНО:
ЗНАЧЕНИЕ ЦЕЛЕВОЙ ПЕРЕМЕННОЙ: *Z* := 4

Некоторые из существующих реализаций более подобны компиляторам, чем интерпретаторам. До начала исполнения они производят предварительный анализ процедур программы с целью образования модулей из машинных кодов, которые включают в себя конкретные шаги унификации, необходимые для результирования процедур с целевым утверждением. Эти модули затем загружаются в память машины вместе со сравнительно простым управляющим модулем, и получающаяся в результате полная объектная программа исполняется намного быстрее, чем в режиме чистой интерпретации.

II.1.4. Вычисления

В дальнейшем вместо термина «логический вывод» мы будем употреблять термин «вычисление». Более точно, *вычислением* (сверху вниз) является каждый вывод, начинающийся с исходного целевого утверждения, который может быть построен при помощи резолюции (сверху вниз) исходя из данной программы. Таким образом, вычисление — это просто последовательность целевых утверждений, каждое из которых, за исключением исходного, порождается в результате применения резолюции к предыдущему целевому утверждению и одной из процедур. Одним из возможных вычислений для программы нахождения следующего элемента будет тогда такое вычисление (G1, G2, G3):

```
G1 : ? след(3,z,1.2.3.4.5.NIL)
G2 : ? след(3,z, 2.3.4.5.NIL)
G3 : ? след(3,z,   3.4.5.NIL)
```

Вычисления можно классифицировать по-разному: как конечные или бесконечные, успешные или неудачные, завершающиеся или незавершающиеся. С указанными классификациями мы еще встретимся дальше в этой главе, а пока достаточно отметить, что в общем случае одна программа может давать несколько различных вычислений. В ходе исполнения программы интерпретатор пытается исследовать *каждое* из них, если, разумеется, ему не дано других указаний. Таким образом, если программа имеет несколько решений, т. е. допускает несколько вычислений, заканчивающихся пустым отрицанием \square , то интерпретатор будет не просто искать все эти решения, а искать каждое из них всеми возможными способами. Это означает, что интерпретатор, как и любая другая программа, выполняющая алгоритм *поиска*, должен сохранять подробный протокол того, что он уже обнаружил, и тех возможностей, которые еще остаются неисследованными. Одна из наиболее приятных особенностей логического программирования, отличающая его почти от

всех остальных программистских формализмов, состоит в том, что это бремя управления поисковыми процессами может быть полностью возложено на интерпретатор.

II. 2. Процедурная интерпретация

Процедурная интерпретация придает операционный характер логическим программам. Суть ее заключается в том, чтобы рассматривать целевые утверждения как множества вызовов процедур, каждый из которых обрабатывается посредством обращения к соответствующей процедуре. В этом отношении процедурная интерпретация очень похожа на процедурные семантики многих традиционных языков программирования. Она позволяет, в частности, смотреть на выполнение логических программ с алгоритмической точки зрения, а не только с точки зрения логического вывода. Понятие процедурной интерпретации является, возможно, наиболее важным достижением в вычислительной логике, которое позволило рассматривать логику как язык программирования.

II.2.1. Структура целевого утверждения

В процедурной интерпретации целевое утверждение

$?A_1, A_2, \dots$ и т. д.

рассматривается как набор *вызовов процедур* A_1, A_2, \dots и т. д. Каждый из этих вызовов указывает *имя* процедуры, являющееся просто предикатным символом. Аргументы предиката называются *фактическими параметрами* вызова. Так, например, целевое утверждение

$?след(3, z, 1.2.3.4.5. NIL)$

содержит только один вызов, который ссылается на процедуру с именем *след* и имеет три фактических параметра: $3, z$ и $1.2.3.4.5. NIL$. Вызов представляет собой некоторую подзадачу, ожидающую решения. (Поэтому вызовы называют иногда «подцелями».) Причина, по которой предикат из целевого утверждения описывается как «вызов», состоит в том, что при его обработке в ходе исполнения программы он вызывает некоторую процедуру с указанным им именем, и она выполняет некоторые вычислительные действия над фактическими параметрами.

II.2.2. Структура процедуры

Процедура представляет собой некоторый способ решения подзадачи. Например, процедура

$след(u, v, w, y) \text{ если } \text{след}(u, v, y)$

может быть прочтана следующим образом: подзадачу вида $\text{след}(u, v, w.y)$ можно решить посредством решения подзадачи $\text{след}(u, v, y)$. Более точно, процедура представляет собой некоторый способ решения вызова. Так, вызов $\text{след}(3, 4, 1.2.3.4.5. NIL)$ может быть решен с помощью приведенной выше процедуры при условии, что в свою очередь может быть решена подзадача $\text{след}(3, 4, 2.3.4.5. NIL)$.

Взаимодействие целевых утверждений, вызовов и процедур станет намного более понятным из последующих разделов. Сейчас же мы введем еще несколько определений.

Предикат, стоящий в процедуре слева от связки *если* называется *заголовком процедуры*. Он дает также *имя* процедуры, которым является его предикатный символ. Аргументы этого предиката называются *формальными параметрами* процедуры. Приведенная выше процедура с именем *след* имеет, стало быть, три таких параметра: *u*, *v* и *w.y*.

Предикаты, стоящие справа от связки *если*, называются *вызовами* и все вместе образуют *тело* процедуры. Эти вызовы представляют собой подзадачи, каждую из которых потребовалось бы решать для того, чтобы процедура решила какой-либо обращающийся к ней вызов. В рассматриваемом примере тело процедуры содержит только один такой вызов.

II.2.3. Операция вызова процедуры

Каждый шаг резолюции, выполняемый в ходе порождения вычисления, можно рассматривать как операцию *вызыва процедуры*. Эта операция включает в себя выбор некоторого вызова из текущего целевого утверждения, выбор процедуры, имя которой совпадает с именем вызова, унификацию фактических и формальных параметров (если это возможно), замену выбранного вызова телом процедуры и, наконец, применение к результату найденного унифициатора. В результате выполнения операции вызова процедуры получается следующее целевое утверждение. Мы говорим тогда, что *вызов был активирован*, а процедура *вызвана*.

Рассмотрим пример, в котором утверждения имеют вид

целевое утверждение G1 : ?A,B

процедура P1 : ?A если C,D

и допустим, что из целевого утверждения выбирается подчеркнутый вызов A. Допустим далее, что вызывается процедура P1, заголовок A, которой унифицируется с выбранным вызовом посредством некоторого унифициатора Θ. Тогда операция вызова процедуры порождает очередное целевое утверждение

G2 : ?CΘ,DΘ,BΘ

Этот шаг можно понимать следующим образом. Как только был определен унификатор Θ , целевое утверждение G1 эффективным образом уточняется до промежуточного утверждения

$$G1' : ?A\Theta, B\Theta$$

Заметим, что каждое решение подзадач $A\Theta$ и $B\Theta$ является также решением более общих подзадач A и B. Например, если вызов A имеет вид $A(x)$, а $\Theta = \{x := f(y)\}$, то $A\Theta$ есть $A(f(y))$; если вызов $A\Theta$ решается, и переменной y в конце концов присваивается в качестве значения терм t , то тем самым решается и более общий вызов $A(x)$, причем ответом будет $x := f(t)$. Точно так же определение унификатора Θ эффективным образом уточняет P1 до процедуры

$$P1' : A\Theta \text{ если } C\Theta, D\Theta$$

которая утверждает, что подзадачу $A\Theta$ можно решить путем решения обеих подзадач $C\Theta$ и $D\Theta$. Поэтому очевидно, что $G1'$ можно решить, решая целевое утверждение

$$G2 : ?C\Theta, D\Theta, B\Theta$$

Процесс замены вызова телом процедуры с учетом тех модификаций, которые необходимы для установления соответствия между фактическими и формальными параметрами, во многом подобен способу определения семантики вызова процедуры для традиционных алголоподобных языков. Единственным отличием является критерий соответствия параметров; в логическом программировании требуется, чтобы параметры были структурно согласованы (унифицируемы), тогда как в других языках накладываются иные условия. Вследствие требования унифицируемости вызов процедуры может оказаться невозможным, даже если ее имя совпадает с именем обращающегося к ней вызова. В случае когда процедура вызывается, т. е. когда ее заголовок унифицируется с вызовом, говорят что процедура отвечает на вызов.

Вспомним, что унификация может также добавлять некоторые присваивания к протоколу связываний порожденного вычисления (содержащему присваивания целевым переменным, которые следует хранить отдельно для последующего извлечения ответа). Допустим, что наши утверждения, записанные полностью, выглядят следующим образом

$$G1 : ?A(x), B(z)$$

$$P1 : A(f(y)) \text{ если } C(y), D(y)$$

Тогда операция вызова процедуры порождает очередное целевое утверждение

$$G2 : ?C(y), D(y), B(z)$$

а унификатор $\Theta = \{x := f(y)\}$ добавляет присваивание $x := f(y)$ к протоколу связываний, где его впоследствии можно будет найти, когда понадобится извлекать ответ, вычисленный в конечном итоге для переменной x . Итак, полностью общее действие операции вызова процедуры можно представить следующим образом:

Текущее целевое утверждение	дают	Следующее целевое утверждение
+		+
Процедура		Присваивания, добавляемые к протоколу связываний

II.2.4. Вход в процедуру и выход из процедуры

Так как на каждом шаге вычисления вызывается некоторая процедура, и в результате этого в целевое утверждение вводятся (в общем случае) новые вызовы (из тела процедуры), исполнение программы можно рассматривать как повторяющийся процесс входа в процедуру и выхода из процедуры. *Вход в процедуру* происходит в тот момент, когда телом процедуры заменяется обращавшийся к ней вызов. *Выход из процедуры* происходит в тот момент, когда оказываются решеными все вызовы из ее тела. (Строго говоря, здесь был описан лишь *успешный* выход из процедуры; с понятием неудачного выхода мы встретимся несколько позже.)

Когда происходит вход в процедуру, вызовы из ее тела, если таковые имеются, называются *непосредственными потомками* обращавшегося к процедуре вызова. Сам этот вызов становится *непосредственно решенным*, если при входе в процедуру у него не возникает непосредственных потомков; такое может произойти только в том случае, когда вызываемая процедура оказывается просто фактом (имеющим пустое тело). В более общей ситуации вызов становится *решенным*, когда либо (i) он становится непосредственно решенным, либо (ii) все его непосредственные потомки становятся решенными.

Эти понятия можно пояснить с помощью следующего примера. (Ради простоты в приводимой ниже программе все параметры опущены.)

```

целевое утверждение G1 : ?A,B
процедуры P1 : A если C,D
P2 : C если F,G
P3: F
P4: G
P5: D
P6: B

```

Исполнение этой программы породило бы тогда следующее вычисление:

вход в P1...	G1 : ?A,B
	G2 : ?C,D,B ...вызов A активирован
	G3 : ?F,G,D,B
	G4 : ?G,D,B
	G5 : ?D,B ...вызов A решен
выход из P1...	G6 : ?B
	G7 : ?(т. е. \square)

Здесь вход в процедуру P1 происходит в тот момент, когда целевое утверждение G2 выводится из G1. В результате этого образуются непосредственные потомки C и D вызова A, обращавшегося к процедуре P1. Вызов F становится непосредственно решенным, когда выводится целевое утверждение G4; точно так же вызов G становится непосредственно решенным, когда выводится G5, и, стало быть, в этот момент становится решенным вызов C, поскольку F и G являются его непосредственными потомками. Когда выводится целевое утверждение G6, непосредственное решение вызова D и полученное ранее решение вызова C приводят к тому, что становится решенным вызов A, и, таким образом, в этот момент происходит выход из процедуры P1. В процессе исполнения программы между входом в процедуру P1 и выходом из нее в разные моменты времени происходили также входы в процедуры P2, P3, P4 и P5 и выходы из них. Наконец, когда выводится целевое утверждение G7, непосредственно решается вызов B; так как вызов A уже был решен, то отсюда вытекает, что исходное целевое утверждение G1 полностью решено, и вычисление успешно завершается пустым целевым утверждением G7. Пустота G7 означает, что больше не осталось никаких вызовов, которые нужно решать с целью решения их предков.

II.2.5. Выбор вызова

В ходе построения вычисления длина текущего целевого утверждения может изменяться в зависимости от числа вызовов, которые вводятся в него в результате обращения к процедурам. Если в текущем целевом утверждении имеется несколько вызовов, то на следующем шаге может быть активирован каждый из них. Принятие решения относительно того, какой именно вызов следует активировать, называется *выбором вызова*; это один из аспектов исполнения логических программ, связанных с необходимостью делать *выбор*.

Действие этого выбора можно увидеть, рассмотрев следующую программу, в которой спрашивается, является ли список

(1, 2, 3) упорядоченным:

```

G1 : ?поряд(1.2.3.NIL)
поряд1 : поряд(NIL)
поряд2 : поряд(u.NIL)
поряд3 : поряд(u.v.y) если u < v, поряд(v.y)
меньш1 : 1 < 2
меньш2 : 2 < 3
```

Допустим теперь, что исполнение этой программы управляетя правилом, согласно которому на каждом шаге активируется первый слева вызов. В этом случае будет получено следующее вычисление (активируемые вызовы здесь подчеркнуты).

G1 : ?поряд(1.2.3.NIL)	
G2 : ? <u>1 < 2</u> ,поряд(2.3.NIL)	в результате вызова
	процедуры поряд3
G3 : ? <u>поряд(2.3.NIL)</u>	в результате вызова
	процедуры меньш1
G4 : ? <u>2 < 3</u> ,поряд(3.NIL)	в результате вызова
	процедуры поряд3
G5 : ? <u>поряд(3.NIL)</u>	в результате вызова
	процедуры меньш2
G6 : ?	в результате вызова
	процедуры поряд2

Фактически это правило предписывает, чтобы проверка упорядоченности пар стоящих рядом элементов списка выполнялась сразу, как только пара появляется в текущем целевом утверждении.

II.2.6. Выбор процедуры

Еще один важный вид выбора, возникающий на каждом шаге исполнения программы, касается выбора вызываемой процедуры. В только что рассмотренной программе об упорядоченном списке не было необходимости в такого рода выборе, поскольку на каждом шаге только одна процедура отвечает на какой бы то ни было активированный вызов. В противоположность этому рассмотрим задачу нахождения элемента z , следующего непосредственно за числом 2 в списке (1, 2, 3, 2, 4). Ее можно сформулировать в виде следующей программы:

```

G1 : ?след(2,z,1.2.3.2.4.NIL)
C1 : след(u,v,u.v.x)
C2 : след(u,v,w.y) если след(u,v,y)
```

Исполнение данной программы должно начинаться с вызова процедуры C2, в результате чего получается целевое утверж-

дение

G2 : ?след(2, z, 2.3.2.4.NIL)

и теперь, поскольку обе процедуры C1 и C2 отвечают на этот вызов, приходится выбирать, к какой из них следует обратиться. Если выбирается C1, то вычисление успешно завершается пустым целевым утверждением

G3 : ?

и при этом извлекается ответ $z := 3$. Если же, напротив, выбирается процедура C2, то получается другое вычисление:

G3' : ?след(2, z, 3.2.4.NIL)

G4 : ?след(2, z, 2.4.NIL)

G5 : ?

и при этом извлекается ответ $z := 4$. Заметим, что целевое утверждение G5 было выведено в результате выбора процедуры C1. Если бы вместо нее была выбрана процедура C2 (а она также отвечает на вызов), то тогда получаемое вычисление не было бы успешным.

Приведенный пример показывает, что выбор процедуры, как и выбор вызова, влияет на ход построения вычисления и может также приводить к порождению различных ответов для целевых переменных. Это важное свойство логических программ, заключающееся в возможности построения целого ряда вычислений, более подробно обсуждается в следующем разделе.

I.3. Пространство вычислений

Читателю должно быть теперь понятно, что исполнение логической программы не обязательно следует только по одному пути вычислений. Существование неоднозначного выбора на различных шагах вычисления означает, что может быть порождено несколько вычислений, дающих, возможно, отличающиеся друг от друга результаты. Сейчас мы переходим к рассмотрению различных видов вычислений, а также условий, влияющих на их порождение во время исполнения программы.

II.3.1. Классификация вычислений

Каждое вычисление, заканчивающееся пустым целевым утверждением (которое обозначается посредством ? или \square), дает некоторое решение исходного целевого утверждения и называется *успешным*.

Если последнее целевое утверждение в вычислении таково, что на некоторый выбранный из него для активации вызов не отвечает ни одна процедура, то вычисление характеризуется как *тупиковое*, и решений оно не дает. В этом случае для наглядности обычно принято добавлять к такому вычислению специальный символ \blacksquare , обозначающий *неудачу*.

Вычисления, которые заканчиваются либо символом \square , либо символом \blacksquare , называются *завершенными*. Вычисления, заканчивающиеся целевыми утверждениями, из которых можно вывести по крайней мере одно следующее целевое утверждение, характеризуются как *незавершенные*; это такие вычисления, которые исполнением программы еще не достроены до одного из *заключений* \square или \blacksquare .

Все рассмотренные выше вычисления являются примерами *конечных вычислений*, и именно с этим видом вычислений программисты обычно хотят иметь дело. Однако некоторые программы приводят к *бесконечным вычислениям*, не дающим никакого заключения.

II.3.2. Полное пространство вычислений

Для каждой данной программы *полным пространством вычислений* называется множество всех вычислений, которые можно получить из этой программы при помощи стандартной операции вызова процедуры. Полное пространство вычислений охватывает все возможные способы выбора вызовов и процедур. Его можно рассматривать как полный перебор различных путей, на которых исполнение программы могло бы искать решения исходного целевого утверждения; некоторые из этих путей могут быть успешными, некоторые — тупиковыми, а некоторые — бесконечными и не дающими результата; одни могут быть эффективными, другие — неэффективными. Каковы бы ни были размеры и структура этого пространства, оно целиком и полностью определяется данной программой и методом построения вывода — резолюцией сверху вниз. Пределы, до которых пространство вычислений исследуется во время исполнения программы, определяются другими факторами, в частности так называемыми *правилами выбора*. Эти правила описываются в следующем разделе.

II.3.3. Действие правил выбора

При *стандартном способе* исполнения программы для активации всегда выбирается первый (самый левый) вызов из целевого утверждения. Действие этого правила, заложенного в интерпретатор, заключается в том, что оно ограничивает исполне-

ние программы поиском только в некотором подмножестве полного пространства вычислений, называемом выбираемым *подпространством*. Более того, все решения программы, если, конечно, они имеются, оказываются вычисляемыми в этом подпространстве, поскольку все остальные успешные вычисления из полного пространства приводят к тому же самому множеству решений, но другими путями, и, стало быть, их удаление из процесса исполнения программы существенного значения не имеет.

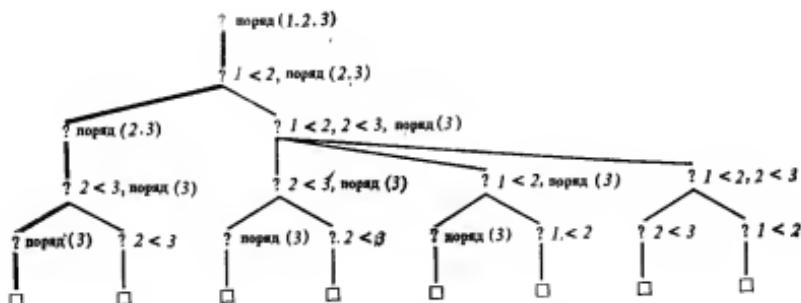


Рис. II. 1. Дерево вычислений для задачи об упорядоченном списке.

Эти свойства выбираемого подпространства иллюстрируются на рис. II. 1, где рассматривается задача об упорядоченном списке, обсуждавшаяся в разделе II. 2.5. Полное пространство вычислений можно изобразить в виде *дерева вычислений*, корнем которого является исходное целевое утверждение, а ветви, выходящие из корня, представляют все различные вычисления. Простоты ради на этом рисунке в термах, служащих именами списков, опущена константа *NIL*. Всего имеется восемь завершенных вычислений, и каждое из них успешное. Они соответствуют восьми различным путям решения указанной задачи при отсутствии ограничений на выбор вызова. Самая левая ветвь, выделенная на рисунке жирными ребрами, представляет единственное вычисление в подпространстве, определяемом стандартным правилом выбора вызова. Вызовы в этом вычислении обрабатываются в следующей последовательности: *поряд(1.2.3.NIL)*, *1 < 2*, *поряд(2.3.NIL)*, *2 < 3* и *поряд(3.NIL)*. В других вычислениях обрабатываются просто разнообразные перестановки этой последовательности вызовов, что соответствует различным правилам выбора. Например, правило, согласно которому всегда выбирается последний (самый правый) вызов из текущего целевого утверждения, ограничивало бы исполнение программы подпространством, представляющим самой правой ветвью дерева.

Вообще говоря, выбираемое подпространство может содержать несколько вычислений, которые отражают альтернативные выборы процедур, отвечающих на один и тот же вызов. (В задаче об упорядоченном списке этого не происходит, поскольку на каждом шаге вызову отвечает только одна процедура.) Таким образом, как только для активации вызовов определено правило выбора, всякое *ветвление* (т. е. наличие нескольких ребер, выходящих из одной вершины дерева), которое еще

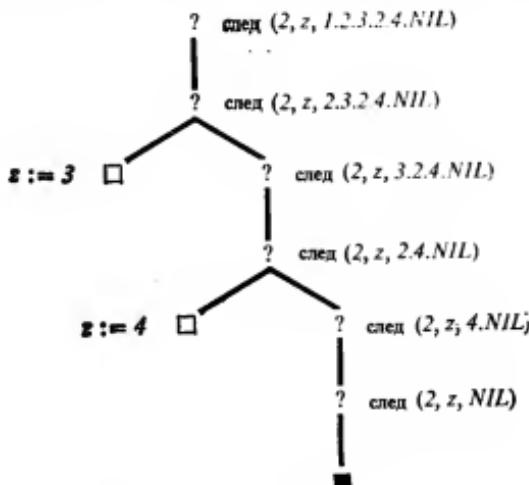


Рис. II. 2. Дерево вычислений для программы нахождения следующего элемента.

остается в *поддереве*, описывающем выбираемое подпространство, полностью определяется существованием выбора между процедурами, отвечающими одному вызову. На рис. II.2 эта ситуация иллюстрируется для задачи нахождения элемента *z*, следующего непосредственно за числом 2 в списке (1, 2, 3, 2, 4), которая решается с помощью программы, приведенной ранее в разделе II.2.6. Здесь, оказывается, полное дерево вычислений и его поддерево, определяемое стандартным правилом выбора вызова, совпадают, поскольку в каждом целевом утверждении имеется только один вызов. Каждая точка ветвления на этом рисунке соответствует возможности выбора между процедурами C1 и C2. Заметим также, что поддерево содержит тупиковое вычисление, которое возникает в том случае, когда предпочтение всегда отдается процедуре C2.

Если бы программа нахождения следующего элемента имела целевое утверждение

`?след(4, z, 1.2.3.2.4.NIL)`

то тогда поддерево содержало бы только одно вычисление, которое заканчивалось бы неудачей, поскольку у данной задачи не существует решения. Говорят, что программа *неразрешима*, если ее подпространство не содержит успешных вычислений; в противном случае программу называют *разрешимой*. Отметим, что неразрешимость программы возникает не только в том случае, когда все вычисления заканчиваются неудачей ■; она возникает тогда, когда всякое вычисление либо заканчивается неудачей ■, либо бесконечно.

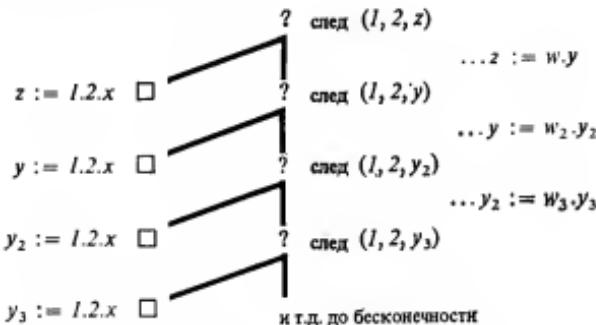


Рис. II. 3. Поддерево, содержащее бесконечное вычисление.

Пример бесконечного вычисления дает еще одна задача о следующем элементе, в которой используются те же самые процедуры, что и прежде, но выбрано другое целевое утверждение:

G1 : ?след(l, 2, z)
 C1 : след(u, v, u.v.x)
 C2 : след(u, v, w.y) если след(u, v, y)

В этой программе ищутся списки, в которых число 2 следует непосредственно за числом 1. На рис. II.3 изображено соответствующее поддерево и приведены присваивания целевым переменным.

Здесь снова точки ветвления появляются тогда, когда вызову отвечают обе процедуры C1 и C2. Все левые ветви завершаются успешно и дают различные ответы:

$z := 1.2.x$
 $z := w.1.2.x$
 $z := w.w_2.1.2.x$
 $z := w.w_2.w_3.1.2.x$
 и т. д.

Каждый ответ определяется лишь частично, поскольку в нем остаются переменные, которым можно придавать произвольные

значения. Переменные w_2 , y_2 , w_3 , y_3 появляются в результате переименования переменных из родительских процедур, необходимого для того, чтобы удовлетворить требованиям стандартной операции вызова процедуры. Заметим, что главная ветвь поддерева является бесконечно длинной; она получается тогда, когда предпочтение все время отдается вызову процедуры C2.

Если предположить, что интерпретатору дано задание исследовать все подпространство целиком, то это, очевидно, приведет к *незавершающемуся исполнению* программы, так как область исследования здесь неограничена. Незавершающееся выполнение может иногда возникать даже в том случае, когда все вычисления в подпространстве конечны, поскольку некоторые программы дают подпространства, имеющие *бесконечное число* таких вычислений. Подобная ситуация может случиться, когда интерпретатор обращается к некоторому встроенному множеству процедур (что это такое, объясняется в разд. II.6), предлагающему бесконечное число различных ответов на вызов и, таким образом, вводящему в поддерево вершины с бесконечным ветвлением.

II. 4. Стандартная стратегия управления

Руководство исполнением логических программ осуществляется главным образом механизмами управления, заложенными в интерпретатор. Программист может оказывать влияние на общий ход исполнения программы, например на последовательность, в которой строятся вычисления, однако более тонкие детали, такие как управление протоколом связываний, могут быть оставлены для разбора интерпретатору.

Предыдущие наши примеры показывали, что логические программы могут давать целое множество вычислений. Это свойство является одним из аспектов *недетерминированности* логических программ. Оно ставит интерпретатор перед необходимостью осуществлять *поиск* для того, чтобы не пропустить ни одного решения. Ниже мы займемся изучением стандартной стратегии, применяемой для управления поиском, и объясним, какое влияние на нее может оказывать программист.

II.4.1. Недетерминированность и поиск

В самом общем смысле недетерминированность означает отсутствие факторов, точно определяющих ход исполнения программы. Логика, следовательно, оказывается недетерминированным языком программирования, ибо в утверждениях логических программ об этом ничего не говорится.

Каждая отдельная программа является *недетерминированной* в том случае, когда она допускает более одного вычисления, т. е. когда ее дерево вычислений содержит ветвления. Такое ветвление вызывается двумя причинами: наличием в целевых утверждениях нескольких вызовов, которые могут быть активированы, и наличием нескольких процедур, способных ответить на один и тот же вызов. Первая причина устраивается за счет введения строгого порядка выбора вызовов, например правила, согласно которому в целевом утверждении всегда выбирается первый слева вызов. Такого рода правило «отрезает» лишние ветви в полном дереве вычислений, оставляя при этом поддерево, содержащее все решения программы. Вторая причина проявляется тогда в наличии какого-либо ветвления, остающегося в этом поддереве; оно и вынуждает предпринимать *поиск* . Поэтому в дальнейшем мы будем называть полученное поддерево *деревом поиска* для данной программы при заданном правиле выбора вызовов. Разумеется, в явном виде до исполнения программы дерева поиска не существует — когда мы говорим о том, что интерпретатор осуществляет поиск в дереве поиска, мы на самом деле имеем в виду, что интерпретатор *строит* его.

II.4.2. Правило вычислений

Правило выбора вызова обычно называют *правилом вычислений*: оно фиксирует множество вычислений, которые будут построены в ходе исполнения программы. Относительные уровни, до которых вычисления строятся интерпретатором во время его движения по дереву поиска, определяют динамические очертания «границы поиска». Вычисления могут строиться по очереди (это, как мы сейчас увидим, является обычным способом исполнения программ), или параллельно, или каким-то иным способом, соединяющим в себе эти две крайности.

Согласно *стандартному правилу вычислений*, на каждом шаге всегда выбирается первый вызов из целевого утверждения. При этом подразумевается, что, когда вызов заменяется на тело вызванной им процедуры, текстуальный порядок вызовов из ее тела не меняется. Получающееся в результате дерево поиска затем, как правило, просматривается по принципу *«сначала в глубину»*. Это значит, что интерпретатор начинает с корня дерева вычислений и потом продвигается вниз по выходящей из корня ветви, строя вычисление сколь угодно долго до тех пор, пока он не достигнет конечной вершины (\square или \blacksquare). До момента окончания этого вычисления никакие другие вычисления строиться не будут.

Если достигается вершина \square , то интерпретатор может извлечь ответ и сообщить о нем устройству вывода. Если же,

напротив, достигается вершина ■, то никакого сообщения о неудаче обычно не выдается. И в том и другом случае могут остаться другие, не просмотренные до сих пор ветви, которые выходят из различных точек ветвления, расположенных выше в дереве поиска. Если это действительно так, то интерпретатором осуществляется *возврат вверх* по только что построенной ветви в поисках ближайшей вершины, из которой выходит еще некоторая неисследованная ветвь¹⁾. Если таковых нет, то дерево поиска построено полностью, и, стало быть, исполнение

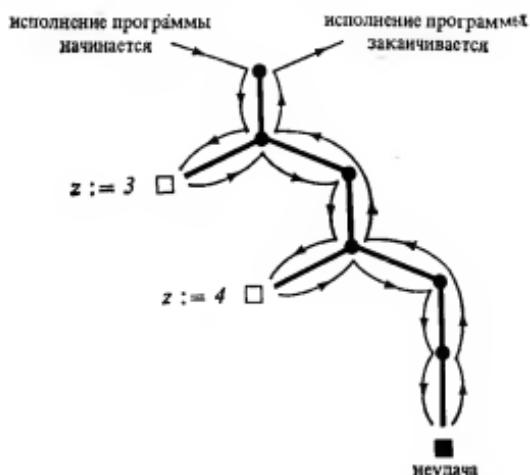


Рис. II. 4. Поиск в глубину с возвратом в полном дереве поиска.

программы заканчивается. В противном случае первая новая альтернативная ветвь, встречающаяся в процессе возврата, продолжается вниз опять по принципу сначала в глубину. Таким способом конечное дерево поиска будет в конце концов полностью просмотрено и будут найдены все решения программы. На рис. II. 4 показано, как осуществляется поиск в глубину в изображенном ранее на рис. II. 2 дереве поиска. Стрелки указывают направление поиска; те из них, которые направлены вверх, указывают на то, что выполняется процесс возврата.

Зная правило вычислений, заложенное в интерпретаторе, программист может принимать решение относительно текстуального упорядочения вызовов в своей программе, таким образом учитывая эффективность ее исполнения. Он может выбирать такие упорядочения, у которых соответствующие подпро-

¹⁾ Такое поведение на англоязычном жаргоне называют *бэктрекингом* (*backtracking*); в дальнейшем мы будем называть его *процессом возврата*. — Прим. перев.

странства могут содержать меньше тупиковых вершин, чем в полном пространстве вычислений, или меньше бесконечных вычислений, или меньше путей, дающих одинаковые решения, или более короткие вычисления. Однако, чтобы точно оценить выгоды того или иного выбора упорядочения, требуется довольно большой опыт работы с имеющейся реализацией языка логического программирования. Возможна, например, такая ситуация, когда для исследования подпространства, содержащего только одно вычисление, потребовалось бы больше времени, чем для исследования другого подпространства той же самой задачи, содержащего несколько вычислений сравнимой длины, поскольку в первом случае интерпретатору могло бы понадобиться больше времени для осознания того, что имеется лишь одно вычисление, чем для обработки более просто ограниченного множества вычислений во втором случае. Это в свою очередь происходит из-за того факта, что возможности для исследования воспринимаются интерпретатором, исходя из числа процедур, потенциально отвечающих каждому вызову, а эффективность их определения зависит от использованных в данной реализации методов хранения, индексирования и выборки процедур.

II.4.3. Правило поиска

Принцип поиска сначала в глубину, применяемый к данному дереву поиска, сам по себе не определяет последовательность, в которой строятся вычисления. Эта последовательность регулируется порядком выбора процедур, отвечающих на вызовы. Всякое правило, задающее указанный порядок, называется *правилом поиска*, поскольку оно определяет направление поиска, избираемое интерпретатором в каждой точке ветвления дерева поиска. Согласно *стандартному правилу поиска*, процедуры, отвечающие на вызовы, выбираются в соответствии с их порядком вхождения в текст программы.

Рассмотрим действие стандартного правила поиска на примере изображенного на рис. II.4 дерева поиска для программы нахождения следующего элемента. Предположим, что процедуры программы расположены в следующем порядке:

C1 : *след(u,v,u.v.x)*
C2 : *след(u,v,w.y)* если *след(u,v,y)*

Когда интерпретатор достигает первой точки ветвления, он вызывает процедуру C1, в результате чего получается решение *z := 3*. Затем он возвращается в ту же точку ветвления и вспоминает, что еще осталась неиспробованной процедура C2, которая также отвечает вызову. Стало быть, следующей вызывается процедура C2, и этот путь вскоре приводит к еще одной точке

ветвления, где применяется все тот же порядок выбора процедур. После того как в конце концов будет найдена тупиковая вершина, в процессе возврата не обнаружится ни одной точки ветвления с еще не исследованными путями. Поэтому траектория управления идет вверх прямо к корню дерева и исполнение программы на этом заканчивается.

Если бы текстуальный порядок процедур C1 и C2 был изменен на обратный, то дерево поиска (определенное только правилом вычислений) осталось бы, конечно, тем же самым. Однако поиск осуществлялся бы тогда в направлении, прямо противоположном тому, которое показано на рис. II.4: сначала встретилась бы тупиковая вершина, затем было бы найдено решение $z := 4$, а уж потом — решение $z := 3$.

Вообще говоря, если предполагается просмотреть все дерево поиска целиком, то упорядочение процедур мало влияет на эффективность исполнения программы. Однако иногда для программиста представляет интерес только некоторая часть дерева поиска. Ему, например, может потребоваться найти лишь одно, не важно какое решение, а не все возможные решения задачи. Поэтому большинство реализаций снабжено различными дополнительными устройствами управления, позволяющими программисту сокращать или как-то иначе модифицировать поисковый процесс. Иногда бывает проще написать программу, допускающую много путей вычислений, и удалить некоторые из них при помощи устройств управления, чем создавать более ограничительную программу, допускающую только один интересующий нас путь. Это лишь одно из многих подобных суждений, оказывающих влияние на *стиль программирования*.

Наиболее известной директивой управления является так называемый *оператор отсечения* (cut operator), представляющий собой просто символ /, который пишется в программе на правах еще одного вызова. При его активации сразу же отсекаются все неисследованные пути, выходящие из всех точек ветвления, которые встретились с момента входа в ту процедуру, чей вызов привел к появлению этого оператора в целевом утверждении. Так, например, если бы мы взяли модифицированную процедуру C1

C1 : след($u, v, u.v.x$) если /

и расположили бы в тексте программы C1 и C2 в указанном выше порядке, то исполнение программы сразу бы закончилось, как только было бы обнаружено решение $z := 3$. В самом деле, активация оператора / отсекла бы оставшуюся возможность выбора процедуры C2 в точке ветвления, удаляя таким образом весь неисследованный остаток дерева поиска.

Часто утверждают, что оператор отсечения является необходимым для практического логического программирования. Несомненно полезный, он тем не менее поощряет «ленивый» стиль программирования: вместо того чтобы тщательно продумывать способ описания отношения, которое действительно требуется, может возникнуть соблазн дать по возможности самое простое описание какого-нибудь более широкого отношения и затем разбросать по тексту программы операторы отсечения, избавляясь тем самым от нежелательных кортежей. Такое «недисциплинированное» употребление оператора отсечения может затруднить анализ и понимание программ.

II.4.4. Стандартная стратегия

Стандартная стратегия, которая управляет процессом поиска, осуществляемым интерпретатором,— это просто сочетание двух правил: стандартного правила вычислений и стандартного правила поиска. Таким образом, основным средством, с помощью которого программист может управлять исполнением своей программы, является выбор текстуального упорядочения процедур и вызовов. Об интерпретаторах, управляемых стандартной стратегией, говорят иногда, что они осуществляют поиск «слева направо в глубину с возвратом», поскольку это название описывает траекторию их движения по дереву поиска, которое нарисовано так, чтобы упорядоченные слева направо ребра дерева, выходящие из каждой точки ветвления, соответствовали текстуальному упорядочению процедур, отвечающих на вызов в данной точке.

Оператор отсечения дополняет стандартную стратегию средством для удаления нежелательных вычислений, и отчасти поэтому интерпретаторы не могут в полной мере использовать полноту резолюции. Другая причина заключается в том, что в соответствии с принципом поиска сначала в глубину исполнение программы может войти в бесконечное вычисление и никогда из него не выбраться, оставляя поэтому часть дерева поиска непросмотренной. В некоторых реализациях имеются средства «контроля зацикливания», которые пытаются распознать бесконечные вычисления, тогда как в более грубых системах может просто накладываться некоторое произвольное ограничение на допустимую длину вычислений. И в том и другом случае посредством искусственного приведения в действие механизма возврата из неугодной вершины дерева вычислений исполнение программы можно освободить от дальнейшего просмотра текущей ветви, хотя при этом пришлось бы пожертвовать полнотой, если бы оказалось, что оставленная ветвь на самом деле успешно завершается.

Довольно часто программисты имеют дело с задачами, требующими исследования только одного пути вычислений, и поэтому им не требуется слишком подробно заниматься вопросом управления. Однако в других ситуациях может потребоваться найти несколько ответов a_1, a_2, \dots и т. д. и тогда можно написать недетерминированную программу, каждое успешное вычисление которой дает один из ответов (как это было в программе о следующем элементе). В этом случае программист может либо осуществлять жесткое управление ходом исполнения программы при помощи механизма текстуального упорядочения и оператора отсечения, либо вообще не беспокоиться об управлении, и тогда самое худшее, что может произойти — это незавершенность исполнения, а менее худшее — его неэффективность. Еще одна возможность заключается в составлении детерминированной программы, вычисляющей единственный ответ, который состоит из терма, объединяющего все a_1, a_2, \dots и т. д. К какой бы из этих методов ни выбирался, он должен в идеале быть направлен на максимальное улучшение логической простоты и ясности программы, а также на повышение продуктивности программирования при условии, разумеется, что затраты на реализацию получаемой в результате программы останутся в приемлемых пределах.

II.5. Поведение программы в ходе вычислений

При наличии интерпретатора, управляемого стандартной стратегией, программист может получать много важных видов вычислительных процессов, просто выбирая подходящие логические структуры для своих входных программ. В этом разделе изучается, как ведут себя логические программы в ходе их исполнения по сравнению с известными алгоритмическими механизмами.

II.5.1. Обработка данных

Большинство программ предназначается для обработки данных. Поэтому имеет смысл выяснить, что означает обработка данных в логическом программировании.

Рассмотрим стандартную операцию вызова процедуры. В общем случае вызов будет содержать несколько переменных, скажем переменные x_1, x_2, \dots и т. д. Заголовок процедуры, к которой этот вызов обращается, также может содержать несколько переменных, скажем y_1, y_2, \dots и т. д. Для наших целей сейчас не играет роли, являются ли сами эти переменные параметрами или они лишь входят в состав параметров.

Во время вызова процедуры соответствующее присваивание данных (унификатор) сопоставляет различные термы некоторым или всем переменным x_i и y_i . Мы интерпретируем каждый такой терм как элемент данных, замечая, что он порождается в результате согласования параметров. Элементы данных, которые присваиваются переменным y_i , называются входными данными, поскольку они вводятся в процедуру из вызова, а элементы данных, которые присваиваются переменным x_i , называются выходными данными, поскольку они выводятся из процедуры в вызов. Эти понятия входных и выходных данных определяются, таким образом, исходя из направления потока данных по отношению к процедуре.

В качестве примера рассмотрим шаг вычисления, на котором процедура C2 вызывается в ответ на первый вызов из целевого утверждения G1:

G1 : ?след(2, z, 1.2.3.2.4.NIL), след(z, 2, 1.2.3.2.4.NIL)
 C2 : след(u, v, w, y) если след(u, v, y)

Отходя ради иллюстрации определений от обычного правила, касающегося присваиваний одним переменным значений других переменных, мы предположим, что используемое здесь присваивание данных таково: $\{u := 2, z := v, w := 1, y := 2.3.2.4.NIL\}$. Термы 2, 1 и 2.3.2.4.NIL — суть элементы данных, которые являются входными относительно переменных u , w и y из процедуры C2, а терм v есть элемент данных, который является выходным по отношению к переменной z из вызова.

Еще одно полезное понятие — это понятие распределения данных. Когда происходит вызов процедуры, входные данные, присвоенные переменной y_i , распределяются по всем вхождениям этой переменной в тело процедуры (т. е. подставляются вместо них). В то же время выходные данные, которые присвоены переменной x_i , распределяются по всем латентным вызовам в целевом утверждении. (Латентными вызовами называются вызовы, еще ожидающие активации: согласно стандартному правилу вычислений, латентными являются все вызовы в текущем целевом утверждении, за исключением первого.) После того как выполнены оба этих распределения данных, следующее целевое утверждение G2 получается путем замены первого вызова в G1 на тело процедуры C2 (см. рис. II. 5).

Распределение данных позволяет по-другому смотреть на применение унификатора после замены вызова. С точки зрения текущего целевого утверждения шаг вычислений передает данные из процедуры через активированный вызов всем латентным вызовам. Это станет более очевидным, если выполнить еще несколько шагов вычисления, предполагая, что используются обыч-

ные процедуры C1 и C2 из программы о следующем элементе.

G1 :	?след(2, z, 1.2.3.2.4.NIL),	след(z, 2, 1.2.3.2.4.NIL)
G2 :	?след(2, v, 2.3.2.4.NIL),	след(v, 2, 1.2.3.2.4.NIL)
G3 :	?	след(3, 2, 1.2.3.2.4.NIL)
G4 :	?	след(3, 2, 2.3.2.4.NIL)
G5 :	?	след(3, 2, 3.2.4.NIL)
G6 :	?	

На рис. II.5 показано, как на первом шаге элемент данных v распределяется по переменным в латентном вызове из G1, и

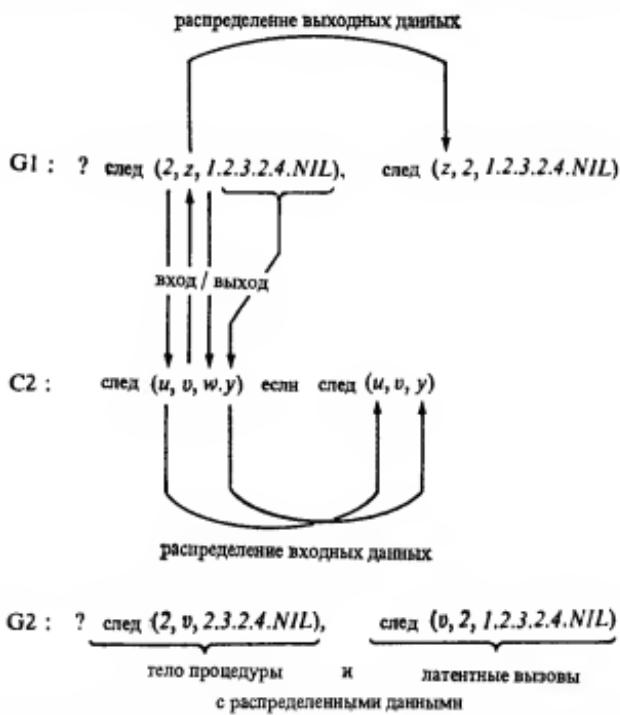


Рис. II.5. Присваивания и распределения данных в операции вызова процедуры.

получается целевое утверждение G2. На втором шаге в результате распределения данных переменная v в латентном вызове из G2 получает значение 3, и мы приходим к целевому утверждению G3. После этого шага в целевом утверждении не осталось ни одной переменной, и, стало быть, распределение выходных данных выполняться больше не будет. Построенное вычисление оказывается успешным, поскольку оно устанавливает, что элемент 3 заключен между двумя элементами 2 в списке (1, 2, 3,

2, 4). Этот ответ извлекается из заключительного состояния протокола связываний, который представляет собой множество $\{z := v, v := 3\}$. Заметим, что протокол связываний можно рассматривать как регистрацию распределений выходных данных, которые происходили во время построения вычисления.

В этом примере все обрабатываемые элементы данных оказываются термами, уже встречавшимися среди фактических или формальных параметров: в ходе вычисления они лишь переставляются с места на место. Однако в других случаях элементы данных могут вычисляться, а не просто выбираться. Рассмотрим вызов процедуры

C1 : след(*u,v,u.v.x*)

в ответ на целевое утверждение ? след(1, 2, z). Присваивание данных $\{u := 1, v := 2, z := 1.2.x\}$ сопоставляет переменной *z* вычисленный терм *1.2.x*, который был построен из входных данных *1* и *2*, а затем передан обратно в виде выходных данных. Следовательно, механизм согласования параметров в интерпретаторе действует как примитивный процессор обработки данных, способный собирать и разбирать структурированные элементы данных в соответствии с образцами из параметров, которые служат, таким образом, в качестве шаблонов.

II.5.2. Упорядочение

Упорядочение, ветвление и итерация являются наиболее известными механизмами, используемыми для организации потока вычислительных событий. *Упорядочение* указывает на выполнение нескольких заданий по очереди в соответствии с точно установленным порядком. Это самый элементарный способ управления почти во всех языках программирования.

В логических программах последовательность выполнения заданий устанавливается просто с помощью записи вызовов в определенном текстуальном порядке. Поэтому, если бы некоторый алгоритм должен был выполнить последовательность действий

считать некоторые входные данные *x*
в результате обработки *x* получить *y*
распечатать выходные данные *y*

целевое утверждение программы можно было бы записать в виде

? считать(*x*), обработать(*x,y*), распечатать(*y*)

Согласно стандартному правилу вычислений, интерпретатор тогда автоматически выполнил бы эти три задания в указанном

порядке. По-другому ту же самую последовательность можно было бы задать внутри некоторой процедуры, такой, например, как

проц(x,y) если считать(x), обработать(x,y), распечатать(y)

В результате обращения к этой процедуре три вызова из ее тела обрабатывались бы по очереди: первый вызов мог бы считать данные для переменной x из некоторого устройства ввода и распределить их во второй вызов; второй вызов в свою очередь мог бы обратиться к некоторой процедуре, выдающей выходные данные для переменной y , и распределить их в третий вызов, который мог бы передать их на устройство вывода. Точные детали работы зависели бы, разумеется, от процедур, предусмотренных для ответа на эти три вызова.

С целью эффективного исполнения программ желательно, как правило, тщательно выбирать подходящую последовательность. Рассмотрим простую задачу, связанную с арифметическим сложением. А именно пусть требуется решить систему уравнений:

$$x + 1 = 3 \quad \text{и} \quad x + y = 3$$

Одним из возможных целевых утверждений, в котором используется предикат **сложить** (x, y, z) для выражения равенства $x + y = z$, является

? сложить($x, 1, 3$), сложить($x, y, 3$)

Отношение **сложить** можно описать множеством процедур, имеющих дело с любым выбранным набором чисел, скажем с числами 0, 1, 2 и 3. Вот одно из возможных множеств

P1 :	сложить(0, z , z)
P2 :	сложить(z , 0, z)
P3 :	сложить(1, 1, 2)
P4 :	сложить(1, 2, 3)
P5 :	сложить(2, 1, 3)

В ходе исполнения программы интерпретатор активирует первый вызов из целевого утверждения и затем последовательно просматривает процедуры P1—P5 в поисках первой из них, отвечающей на активированный вызов. Такой процедурой оказывается P5. Вызывая ее, интерпретатор присваивает переменной x значение 2 и получает следующее целевое утверждение:

? сложить(2, y , 3)

Теперь активируется этот вызов, и снова просматриваются процедуры P1—P5. Первой отвечающей на вызов процедурой вновь оказывается P5, и в результате обращения к ней переменной y

будет присвоено значение 1. Вычисление успешно завершается и дает решение $\{x := 2, y := 1\}$ системы уравнений. Это довольно эффективный способ решения нашей задачи.

Допустим теперь, что в исходном целевом утверждении вызовы расположены в обратном порядке:

G1 : ?сложить(x,y,3),сложить(x,1,3)

В результате этого получается намного более сложное исполнение программы. Оно выглядит следующим образом:

G2 : ?сложить(0,1,3) посредством вызова P1; это целевое утверждение оказывается неудачным, и поэтому происходит возврат к G1
 G2a : ?сложить(3,1,3) посредством вызова P2; это целевое утверждение оказывается неудачным, и поэтому происходит возврат к G1
 G2b : ?сложить(1,1,3) посредством вызова P4; это целевое утверждение оказывается неудачным, и поэтому происходит возврат к G1
 G2c : ?сложить(2,1,3) посредством вызова P5; это целевое утверждение оказывается успешным
 G3 : ? задача решена, ответ $\{x := 2, y := 1\}$

Хотя это исполнение, очевидно, менее эффективно, правильный ответ все равно получен. Говорят, что второе исполнение программы является «менее детерминированным», чем первое, поскольку в нем потребовалось исследовать большее количество вычислений, причем почти все они заканчивались неудачей. Изменение последовательности вызовов эквивалентно изменению правила вычислений, которое в свою очередь меняет выбираемое дерево поиска, предъявляемое интерпретатору.

Как правило, лучше всего стремиться к наиболее детерминированному поведению, хотя это не всегда оказывается самой лучшей политикой: многое зависит от таких особенностей различных деревьев поиска, как распределение тупиковых вершин и сложность протоколов связываний. Часто деревья поиска с наименьшей степенью ветвления возникают в том случае, когда в последовательностях вызовов минимизируется число переменных, встречающихся в вызове на момент его активации. Именно так обстоит дело в только что рассмотренном примере: в первом исполнении активируются вызовы $\text{сложить}(x, 1, 3)$ и $\text{сложить}(2, y, 3)$, каждый из которых содержит только по одной переменной, в то время как во втором исполнении активируется вызов $\text{сложить}(x, y, 3)$, содержащий уже две переменные. В общем случае чем больше переменных содержит активируемый вызов, тем большее число процедур может ответить на него, и,

следовательно, тем большая степень ветвления будет у дерева поиска.

И наконец, отметим важное отличие логического программирования от традиционного, в котором нельзя произвольным образом изменять последовательность вызовов процедур, поскольку каждая процедура логически зависит, как правило, от предположения, что некоторым передаваемым ей параметрам уже будут присвоены какие-то данные в результате предшествующих событий. К логическим же процедурам, таким как P_1 , может обращаться вызов $\text{сложить}(0, I, I)$, фактические параметры которого не содержат переменных, или вызов $\text{сложить}(x, y, I)$, где переменным x и y еще не присвоено никаких значений. Это свойство логических процедур, отсутствующее почти во всех других языках программирования, называется *недетерминированностью входа-выхода* или *инвертируемостью*. Суть его в том, что сама по себе процедура не определяет полностью, какие из ее формальных параметров получают входные данные из вызова, а какие возвращают ему выходные данные: напротив, их статус по отношению к входу и выходу частично определяется в соответствии с вызовом, который обращается к этой процедуре.

II.5.3. Ветвление

Ветвление указывает на обстоятельства, в силу которых путь вычисления достигает точки (точки ветвления), откуда дальнейшие вычисления могут развиваться в любом из нескольких направлений. Ветвление обычно рассматривают на основе *тестирования*, т. е. проверки в точке ветвления выполнения некоторых условий с тем, чтобы решить, в каком из направлений двигаться дальше.

Существует несколько способов для достижения такого поведения в логических программах. Все они основываются, в конечном счете, на использовании нескольких процедур с одним и тем же именем: множество процедур можно рассматривать как единый объект, к которому обращается вызов, но который предлагает при входе в него несколько различных внутренних траекторий. Предположим, что нам требуется выполнить некоторое задание на объектах x , y и z , причем предпринимаемые действия зависят от состояния еще одного объекта u . Более точно задание формулируется следующим образом:

```
если  $u = 1$  то      выполнить действие-1 на объектах  $x,y,z$ 
иначе если  $u = 2$  то  выполнить действие-2 на объектах  $x,y,z$ 
иначе никаких действий не выполнять
```

Это задание можно выразить на языке логики при помощи двух процедур с одинаковым именем задание:

T1 : задание($1, x, y, z$) если действие-1(x, y, z)
T2 : задание($2, x, y, z$) если действие-2(x, y, z)

При активации вызова вида задание(u, x, y, z) последующее поведение зависит от состояния параметра u в момент активации. Если этот параметр является константой 1, то на вызов отвечает только процедура T1, и поэтому будет выполнено действие-1; точно так же, если этот параметр есть константа 2, то на вызов отвечает только T2, и, стало быть, будет выполнено действие-2. Если же u является каким-либо другим термом, отличным от переменной, то ни одна из данных процедур на вызов не отвечает и поэтому никаких действий не выполняется. Следовательно, рассматриваемое множество процедур достигает желаемого поведения, связанного с проверкой условия и ветвлением, когда в точке вызова дискриминатор u уже определен.

Поведение будет совершенно иным, если при активации вызова задание значение дискриминатора u еще не установлено, ибо когда u есть просто переменная, на вызов отвечают обе процедуры T1 и T2. В этой ситуации по очереди будут выполнены действие-1 и действие-2, поскольку дерево поиска содержит узел ветвления, все выходы из которого автоматически исследуются интерпретатором.

Показанное в этом примере тестирование является по существу «тестированием по типу»: проверка заключается в том, чтобы определить, принадлежат ли фактический и формальный параметры одному и тому же структурному типу, т. е. будут ли они унифицируемы. Вызов процедуры, управляемый такого рода тестированием, обычно называют «вызовом по образцу»; он используется помимо логики еще в нескольких других языках программирования (например, в языке PLANNER).

Когда программист нуждается в более сложной проверке для управления выбором действия, то эту проверку можно поместить в качестве вызова в процедуры, в которых рассматриваются отдельные возможные действия. Например,

T1a : задание(u, x, y, z) если решить($u, 1$), действие-1(x, y, z)
T2a : задание(u, x, y, z) если решить($u, 2$), действие-2(x, y, z)

Здесь предикат решить(u, k) используется для выражения того, что k есть результат (1 или 2), полученный в соответствии с данными, которые были присвоены переменной u ; эти данные могли бы быть сколь угодно сложными, точнее, настолько сложными, насколько это позволяют процедуры решить, предусмотренные для их анализа.

Тестирование и ветвление, очевидно, не являются какими-то новыми возможностями, добавляемыми к логической формальной системе программирования; наоборот, они оказываются внутренне присущими ей свойствами. Каждая операция вызова процедуры включает в себя проверку соответствия параметров, и каждый вызов можно рассматривать как тест, проверяющий, удовлетворяют ли какие-либо данные некоторому свойству, например, будет ли элемент данных i требовать результата k . Используя эти механизмы, в языке логики можно сформулировать все виды принятия алгоритмических решений.

II.5.4. Итерация

Итерацию обычно рассматривают как процесс неоднократного выполнения какого-то сегмента программы; при каждом повторении, или «шаге итерации», вычисление, предписанное этим сегментом, выполняется полностью, и только затем начинается следующий шаг итерации. В логическом программировании имеются два основных способа достижения такого поведения.

В одном из них используется свойственное интерпретатору итеративное поведение, когда он просматривает серию процедур в поисках той, которая отвечает на вызов. Простой пример дает задача поиска некоторого конкретного элемента в заданном списке L . Пусть список $L = (3, 4, 9, 5)$ представлен следующими процедурами M1 — M4:

$$\begin{aligned} M1 &: \exists(3, 1, L) \\ M2 &: \exists(4, 2, L) \\ M3 &: \exists(9, 3, L) \\ M4 &: \exists(5, 4, L) \end{aligned}$$

Здесь (как и в примере 1 из разд. I.6) предикат $\exists(i, i, x)$ означает, что элемент i занимает в списке x позицию с номером i . Задача состоит в том, чтобы найти, какую позицию i (если вообще таковая имеется) занимает в L какой-либо заданный элемент, скажем, 9. Стало быть, подходящим целевым утверждением является

$$\exists(9, i, L)$$

В ходе исполнения программы интерпретатор в поисках отвечающей на этот вызов процедуры последовательно сравнивает число 9 с элементами списка L ; эти сравнения представляют собой часть механизма унификации. Когда будет найдена процедура M3, вычисление успешно завершается и дает ответ $i := 3$. Каждый шаг в просмотре заголовков процедур M1 — M4 образует шаг итерации в итеративном процессе, который возникает благодаря заложенной в интерпретаторе стратегии управления.

Заметим, что программисту не нужно было предусматривать никаких конструкций, определяющих, как итерация должна начинаться, продолжаться или заканчиваться.

Другой способ получения итеративного поведения заключается в использовании *итеративной процедуры*. Это такая процедура, в теле которой содержится в точности один вызов с тем же именем, что и у заголовка процедуры, причем этот вызов должен стоять в ее теле на последнем месте. Таким образом, итеративная процедура имеет либо вид

A если A'

либо

A если B₁, ..., B_m, A'

где **A** и **A'** содержат одинаковые имена процедур, отличающиеся от имен в вызовах **B₁**, ..., **B_m**.

Итеративную процедуру можно использовать для решения задачи нахождения суммы *y* всех чисел, являющихся элементами списка *L*. На каждом *i*-м шаге итерации к уже накопленной сумме *x* (изначально она полагается равной 0) прибавляется *i*-й элемент *L*. Когда будут выполнены *n* шагов, где *n* — это длина списка *L*, последнее значение *x* и будет искомым ответом *y*. Всю программу можно структурировать тогда следующим образом

G1 : ?суммировать(0, y, I, 4, L)

A1 : суммировать(y, y, i, n, L) если i > n

A2 : суммировать(x, y, i, n, L) если i ≤ n, э(u, i, L), сложить(x, u, x'), сложить(i, I, i'), суммировать(x', y, i', n, L)

а также процедуры *M1—M4*, определяющие список *L*, и процедуры для отношений *>*, *≤* и *сложить*.

Как обычно, предикат *сложить* (*u, v, w*) означает, что *u + v = w*. Предикат *суммировать* (*x, y, i, n, L*) означает, что *x* есть сумма всех элементов *L*, предшествующих *i*-му элементу, а *y* есть результат сложения *x* с суммой всех элементов с *i*-го до *n*-го включительно.

Процедура *A2* является итеративной процедурой, и при обращении к ней выполняется следующий общий шаг итерации:

проверить неравенство *i ≤ n*

выбрать элемент *u*, стоящий на *i*-й позиции в *L*

сложить *u* с *x*, получая *x'* (обновление суммы)

сложить *i* с *I*, получая *i'* (обновление счетчика шагов)

перейти к следующему шагу итерации

Процедура *A1* лишь завершает итерацию, когда число шагов *i* превзойдет *n*, и окончательный ответ *y*, стоящий на второй позиции аргументов, сделается равным накопленной сумме,

стоящей на первой позиции. Исходное целевое утверждение G1 вводят в вычисление необходимые начальные значения накопленной суммы (O) и счетчика шагов (I). Получаемое в результате поведение показано ниже. Здесь ради краткости мы опустили ряд промежуточных целевых утверждений; их восстановление послужит хорошим упражнением для читателя. Иллюстрируемые целевые утверждения G1, G6, G11 и т. д. появляются в те моменты, когда инициируются новые шаги итерации. На каждом таком шаге сначала, согласно стандартному правилу поиска, испытывается процедура A1, которая в случае $i \leq n$ оказывается неудачной — и тогда вместо нее испытывается процедура A2. Дерево поиска содержит поэтому несколько коротких тупиковых ветвей, соответствующих повторным проверкам счетчика i . Вычисление G1 — G23 показывает только успешный путь в дереве поиска.

G1 : ? суммировать(0, y, 1, 4, L)

G2 : ? $I \leq 4$, $\exists(u, I, L)$, сложить($0, u, x'$), сложить($1, 1, i'$),

суммировать($x', y, i', 4, L$)

1

G6 : ? суммировать(3, y, 2, 4, L)

1

3

G11 : ? Суммировать(7,и,3,4,L)

3

1

С16 : ? суммировать(16,и,4,4,1)

1

1

G21 : ? суммировать(2I,y,5,4,L)

G22 : ? 5 > 4

G23 : ? задача решена, ответ $y := 21$

Отметим, что в логическом программировании можно достигать такого алгоритмического поведения, которое лучше всего, по-видимому, было бы назвать «недетерминированной итерацией». Оно возникает тогда, когда или вызов **A** или какие-либо из промежуточных вызовов **B₁**, ..., **B_m** в общей итеративной процедуре могут быть решены несколькими способами. В этом случае после каждого найденного решения итерация «прокручивалась» бы заново (посредством механизма возврата) для того, чтобы исследовать другие альтернативы. В рассмотренном примере этого не происходит, поскольку каждый из вызовов суммировать, \leq , $>$, \neq и сложить решается только одним спо-

собом; мы имеем лишь обычный, детерминированный тип итерации.

Для закрепления понятия итеративной процедуры мы приведем здесь еще один короткий пример. Метод Ньютона приближенного вычисления квадратного корня из числа x начинается с выбора начального приближения y . Это приближение считается достаточно точным, если $|x - y^2| \leq \text{err}$, где err — некоторая заранее определенная погрешность. В противном случае более точным приближением будет $\frac{1}{2}(y + x/y)$, которое можно вычислить и точно так же подвергнуть проверке на погрешность. Таким образом, в этом методе итеративно порождается последовательность приближений, сходящаяся к точному значению \sqrt{x} .

В следующей программе, предназначеннной для вычисления $\sqrt{5}$ при начальном приближении 1, предикат **ньютон** (x, y, z, err) используется для выражения того, что z — приблизительное значение \sqrt{x} с погрешностью err , получаемое в результате построения последовательности $(y, \frac{1}{2}(y + x/y), \dots$ и т. д.).

```
G1 : ? ньютон(5, 1, z, 0.05) (выбранная погрешность err = 0.05)
N1 : ньютон(x, z, z, err) если умножить(z, z, zsq), аbs(x, zsq, diff),
    diff ≤ err
N2 : ньютон(x, y, z, err) если разделить(x, y, ratio),
    сложить(y, ratio, sum), разделить(sum, 2, y'),
    ньютон(x, y', z, err)
```

а также процедуры для предикатов **умножить**, **разделить**, **сложить**, **абс** и \leq .

Арифметические предикаты имеют следующий смысл:

абс(u, v, w)	означает, что $ u - v = w$
сложить(u, v, w)	означает, что $u + v = w$
умножить(u, v, w)	означает, что $u * v = w$
разделить(u, v, w)	означает, что $u / v = w$

Первая процедура **ニュтона** включает в себя проверку точности приближения; если она оказывается неудачной, то вторая процедура инициирует шаг итерации. В результате исполнения программы получается следующее успешное вычисление:

```
? ньютон(5, 1, z, 0.05)
```

```
·  
·
```

```
? ньютон(5,3,z, 0.05)
:
:
?
? ньюトン(5,2.333,z, 0.05)
:
:
?
? ньютон(5,2.238,z, 0.05)
:
:
?
0.009 ≤ 0.05
?      задача решена, ответ z:= 2.238
```

II.5.5. Рекурсия

Рекурсией называется поведение, которое получается, когда некоторый сегмент программы обращается сам к себе до полного завершения вычислений, соответствующих этому сегменту. В логическом программировании такое поведение обычно достигается за счет использования *рекурсивной процедуры*, т. е. процедуры, имя которой встречается по крайней мере в одном вызове из ее тела. (Итеративные процедуры являются, следовательно, частными случаями рекурсивных процедур.)

Для иллюстрации рекурсии часто используется задача вычисления факториала. Пусть предикаты **факториал**(*u, v*), **вычесть**(*u, v, w*) и **умножить**(*u, v, w*) означают, что $u! = v$, $u - v = w$ и $u * v = w$ соответственно. В следующей программе, вычисляющей значение $3!$, содержится рекурсивная процедура F2. Процедура F1, которая завершает рекурсивный процесс, называется обычно *базисной* процедурой.

```
G1 : ? факториал(3,z)
F1 :   факториал(0,1)
F2 :   факториал(x,y) если x > 0, вычесть(x,1,x'),
          факториал(x',y'), умножить(x,y',y)
```

Для того чтобы вычислить $3!$, программа вычисляет $2!$ и затем умножает результат на 3 . Точно так же при вычислении значения $2!$ умножение $1!$ на 2 остается незаконченным, пока вычисляется $1!$. Таким образом, в текущем целевом утверждении будут до тех пор накапливаться латентные вызовы процедуры **умножить**, которые представляют ожидающие вычисления умножения, пока в конце концов вызов, требующий вычислить $0!$, не будет непосредственно решен с помощью процедуры F1. Ниже приводится схема полученного рекурсивного вычисления. Не-

которые из промежуточных целевых утверждений в ней опущены.

? факториал($3, z$)

.

? факториал($2, y'$), умножить($3, y', z$)

.

? факториал($1, y''$), умножить($2, y'', y'$), умножить($3, y', z$)

.

? факториал($0, y'''$), умножить($1, y''', y''$), умножить($2, y'', y'$),
умножить($3, y', z$)

. Текущий вызов является процедурой F1, присваивающая
переменной y''' значение 1, и затем по очереди
решаются латентные вызовы умножить

? умножить($3, 2, z$)

? задача решена, ответ $z := 6$

Рекурсивные программы часто дают компактные и элегантные описания интересующих нас отношений, однако недостаток их состоит в том, что во время счета по таким программам образуется увеличивающийся в размерах стек латентных вызовов, для хранения которого может потребоваться неприемлемый объем памяти. Как правило, рекурсии пользуются тогда, когда либо эти затраты памяти компенсируются элегантностью программы, либо не имеется практических нерекурсивных алгоритмов для решения поставленной задачи.

II.6. Встроенные средства

Для того чтобы избавить программиста от неудобств, связанных с определением элементарных и часто используемых операций, большинство языков программирования снабжено рядом процедур и функций, имеющих фиксированное, внутренне определенное значение. В математической логике предикатные и функциональные символы не обладают раз и навсегда фиксированными значениями; те значения, которые они получают, полностью определяются содержащими их программами. Тем не менее в каждой конкретной реализации логики как языка программирования некоторым из этих символов могут быть приданы фиксированные значения; и действительно, практически все существующие на сегодняшний день реализации снабжены значительным числом таких предикатных и функциональных символов.

II.6.1. Встроенные процедуры

Наиболее распространенными видами элементарных вызовов процедур являются, по-видимому, те, которые осуществляют проверку равенства и относительной величины. Поэтому во многих интерпретаторах фиксируются значения таких предикатных символов, как

$$= \neq < \leq > \geq$$

в соответствии с их обычным математическим смыслом при применении к действительным и целым числам. Таким образом, программисту, пишущему процедуру

поряд (*u.v.x*) если *u < v*, поряд(*v.x*)

не требуется предусматривать дополнительные процедуры для отношения *<*. Когда активируется вызов типа *1 < 2*, интерпретатор просто обращается к внутреннему встроенному тесту, решающему этот вызов непосредственно. Получаемый при этом результат будет точно таким, как если бы программист написал свои собственные процедуры

$$\begin{aligned} 0 &< 1 \\ 1 &< 2 \\ 0 &< 2 \\ \text{и т. д.} \end{aligned}$$

Такого рода средства должны быть в состоянии оперировать с вызовами, имеющими какие угодно параметры. Так, например, в результате решения вызова *2 < x*, спрашивающего, какие числа следуют за числом 2, должны быть выданы соответствующие выходные данные для параметра *x*. Поскольку на этот вопрос имеется бесконечно много ответов, интерпретатор должен выдавать их последовательно (как будто бы он последовательно испытывал каждую из неявно предполагаемых процедур, отвечающих на данный вызов) согласно некоторому внутренне определенному упорядочению. Следовательно, в результате выполнения целевого утверждения

$$? 2 < x, x < 6$$

ответы могли бы быть выданы в таком порядке: *x := 3, x := 4* и *x := 5*.

Будет ли интерпретатор продолжать и дальше порождение бесконечного числа все больших значений *x* из первого вызова, ни одно из которых не способно решить второй вызов, зависит от свойств конкретной реализации. Такое поведение, во всяком случае, можно было бы предотвратить, правда довольно неуклю-

же, переформулировав задачу несколько по-другому

```
? 2 < x, x < 6, проверить(x)
    проверить(x) если x ≠ 5
    проверить(5) если /
```

Тогда как только значение 5 будет передано вызову проверить, активация оператора / отсечет все неиспользованные еще возможности, возникающие из вызова $2 < x$.

Соответствующие библиотеки встроенных процедур будут, естественно, варьироваться в зависимости от имеющейся в виду области применений. Реализация, специально приспособленная для математической работы, могла бы предоставить средства для обработки векторов, матриц, полиномов и т. д., тогда как в реализации, предназначеннной для работы с базами данных, потребовались бы совершенно иные средства. В этой книге мы предполагаем, что фиксированные значения имеют только шесть приведенных выше предикатных символов.

II.6.2. Встроенные функции

Почти для всех приложений, связанных с вычислениями, требуются элементарные арифметические операции. Поэтому в ряде интерпретаторов фиксированные значения сопоставлены некоторым предикатным символам, таким как сложить, умножить и т. д. В других же интерпретаторах фиксированные значения могут быть приданы функциональным символам, подобным + и *, в результате чего получаются наиболее компактные программы.

Рассмотрим в качестве примера следующую новую формулировку задачи вычисления факториала

```
? факториал(3, z)
    факториал(0, 1)
    факториал(x, y) если x > 0, факториал(x - 1, y'), y = x * y'
```

Если бы функциональным символам — и * здесь не было придано никакого специального значения, то в ходе исполнения программным переменным присваивались бы в качестве значений громоздкие термы вида $3 - 1$ и $3 - 1 - 1$. Более того, в конце концов был бы активирован вызов $\text{факториал}(3 - 1 - 1 - 1, y'')$, который не смог бы обратиться к первой процедуре факториал , поскольку термы $3 - 1 - 1 - 1$ и 0 не унифицируемы. Мы, очевидно, имеем в виду, что оба этих терма представляют число «нуль», но интерпретатору об этом может быть ничего не известно.

Для того чтобы добиться разумного арифметического поведения от такого рода программ, функциональным символам вида 2 и * можно придать в интерпретаторе фиксированные

значения. Тогда в момент активации вызова с фактическим параметром, подобным терму $3 - 1$, интерпретатор сможет сразу вычислить этот терм в арифметическом смысле, заменив его на единственную константу, в данном случае на константу 2. При такой организации, следовательно, константам $0, 1, 2, \dots$ и т. д. также сопоставляются специальные значения: они рассматриваются как обычные натуральные числа, а не как произвольные неинтерпретированные символы. Это не нарушает логический базис формальной системы при условии, что символы $0, 1, 2, \dots$ и т. д. обрабатываются непротиворечивым образом, так, будто они являются только сокращениями, построенными интерпретатором вместо более сложных термов.

С помощью соответствующих расширений основного механизма унификации могут быть реализованы также более сложные операции, такие, например, которые способны согласовать, скажем, термы b и $3 * x$, получая в результате присваивание $x := 2$. При этом, однако, требуется аккуратное семантическое обоснование, гарантирующее, что исполнение останется по-прежнему эквивалентным логическому выводу. На протяжении этой книги мы будем использовать только наиболее безобидные способы употребления встроенных функций, свободно применяя стандартные арифметические операторы, но разрешая производить арифметические вычисления только для термов без переменных. Программа вычисления факториала удовлетворяет этому ограничению. Она может обрабатываться следующим образом:

```
? факториал(3, z)
? факториал(3 - 1, y'),      z = 3 * y',
? факториал(2 - 1, y''),     y'' = 2 * y'',      z = 3 * y'
? факториал(1 - 1, y'''),    y''' = 1 * y''',      y' = 2 * y'',      z = 3 * y'
?                                     y''' = 1 * 1,      y' = 2 * y'',      z = 3 * y'
?                                     y' = 2 * 1,      z = 3 * y'
?                                     z = 3 * 2
? задача решена, ответ z := 6
```

В приведенном вычислении всякий раз, когда активируется какой-либо вызов, содержащий арифметическое выражение без переменных, это выражение вычисляется и заменяется на его значение; вызов процедуры происходит дальше обычным образом.

II.7. Исторический очерк

Логическое программирование возникло главным образом благодаря успехам в автоматическом доказательстве теорем, в частности благодаря разработке принципа резолюции. Одно

из первых исследований, связывающих резолюцию с программированием для ЭВМ, было предпринято Грином (1969), который показал, что механизм извлечения ответа можно использовать для синтеза традиционных программ путем применения резолюции к их спецификациям, выраженным в логике дизъюнктов. Синтезаторы, которые были разработаны для этой цели, можно было бы считать предшественниками современных логических интерпретаторов.

Общая идея, состоящая в рассмотрении логических предложений как операторов в программах, а управляемого вывода — как исполнения программ, была исследована Хайсом (1973), Сандвеллом (1973) и другими. Однако осознанию того, что логика является исполняемым языком программирования, в особенности способствовала процедурная интерпретация, сформулированная Ковальским (1974б). Это было существенное продвижение вперед, необходимое для адаптации понятий из области автоматического доказательства теорем к методам вычислений, уже понятным программистам. Превосходное краткое изложение процедурной интерпретации можно найти в статье ван Эмдена (1977а).

Успехи в технологии реализации также в значительной мере способствовали представлению логики как практической формальной системы программирования. Первый экспериментальный интерпретатор был реализован Русслем, Колмероэ и другими в университете Экс — Марсель в 1972 г. Ему было дано имя Пролог («программирование на языке логики» — Programming in Logic), и он оказал сильное влияние на разработку последующих систем. Вслед за этим важным первым шагом более практические реализации были разработаны Баттани и Мелони (1973), Бранохе (1976), Уорреном (1977а) и Робертом (1977). С тех пор различные реализации Пролога получили очень значительное распространение. Они покрывают широкий диапазон философий проектирования, областей применения и вычислительных машин, на которых выполнены реализации.

Термины «логическое программирование» и «программирование на языке Пролог» часто употребляются как равнозначные, однако подразумеваемая стратегия управления в Прологе, в роли которой выступает определенная ранее в этой главе стандартная стратегия, отнюдь не является единственной стратегией, имеющейся для исполнения логических программ. Совершенно иная стратегия лежит, например, в основе разработанной Ковальским (1975) процедуры доказательства с помощью «графа связей». Эта процедура действует, используя специальную схему активации ребер, которая применяется к ребрам

графа, связывающего вызовы с отвечающими на них процедурами. Самая первая попытка реализовать данный метод, была, по-видимому, предпринята Тернлундом (1975а).

Во всяком случае важно понимать, что даже Пролог-системы значительно отличаются друг от друга из-за наличия в них различных дополнительных средств, предусмотренных для обогащения ресурсов программиста. В большинстве интерпретаторов, например, допускаются разнообразные способы модификации стратегии управления. Иногда такие усиления стандартной стратегии приводят к вычислениям, которые нельзя полностью обосновать, исходя только из логического вывода, и в этом случае интерпретаторы называют потенциально «нечистыми»; примером подобного интерпретатора является первоначальный Пролог, реализованный в Марселе. В противном случае, когда интерпретатор всегда ведет себя в соответствии со строгим логическим выводом (основанным, быть может, и не на резолюции), его называют «чистым»; система IC-Пролога в лондонском Имперском колледже, написанная Кларком и Маккейбом (1979а), является чистой, несмотря на то что в ней имеется несколько довольно сложных механизмов, дополняющих стандартную стратегию. Подробное учебное описание практического программирования с помощью такого рода Пролог-систем, разработанных Уорреном (1977а, 1979), а также Клоксином и Меллишем (1980), дается в книге, где впервые специально рассматривается логическое программирование; написанная Клоксином и Меллишем (1981), она ориентирована главным образом на реализацию Пролога на машине DEC-10.

В статье Ковальского (1981б) обсуждается, до какой степени язык Пролог достигает цели в реализации общей концепции логики как языка программирования, и где его постигла неудача. Хотя Пролог возникает как новый способ понимания программирования, тем не менее стоит помнить и о том, что предшествовали Прологу новые пути в понимании математической логики. К ключевым достижениям здесь Ковальский относит, во-первых, осознание того, что логика обладает как прагматическим, так и семантическим содержанием (благодаря процедурной интерпретации), и, во-вторых, осознание того, что вывод можно сделать целенаправленным (посредством систем, подобных резолюции) вопреки традиционной его репутации как ориентированного главным образом на получение следствий. Введения как в логическое программирование, так и в Пролог были написаны недавно Ковальским (1981б, 1983б), а также Саммутом и Саммутом (1983а, б); кроме того, Кларком и Тернлундом (1982) и Уорреном и ван Канагеном (1985) были изданы сборники передовых исследовательских статей. Хорошее

освещение современного состояния исследований в этой области дается в материалах семинара по логическому программированию (1983), симпозиума в г. Атлантик-Сити (1984) и предстоящей II Международной конференции по логическому программированию в Упсале (1984). В Сиракьюсском университете (Нью-Йорк) готовится новый журнал логического программирования (*Journal of Logis Programming*) под редакцией Дж. А. Робинсона¹¹. Подробная история возникновения логического программирования изложена в статье Ковальского (1984).

¹¹ Этот журнал издается с 1984 г. — Прим. перев.

III. Стиль программирования

Хороший «стиль программирования» улучшает общее качество программы и оказывается столь же важным для успешного логического программирования, сколь и для традиционного программирования. В написанной в хорошем стиле программе допущения относительно решаемой задачи должны быть отчетливо воспринимаемы, и в то же время программа должна представлять собой пригодный для вычислений алгоритм.

В традиционном программировании эти две последние цели трудно согласовать между собой, что хорошо известно студентам, изучающим «структурное программирование». Главная трудность здесь — решить, как описать детали управления исполнением программы с тем, чтобы гарантировать эффективность и не затуманить при этом ее логические цели, поскольку два этих аспекта программы являются взаимозависимыми и могут предъявлять противоречия друг другу требования.

Быть может, наиболее существенное преимущество, достигаемое за счет использования логики в качестве языка программирования, заключается в том, что дескриптивное содержание логической программы не зависит от каких-либо допущений относительно механизма исполнения. Вследствие этого оказывается возможным разрабатывать атрибуты программы, связанные с ее логикой и поведением в период исполнения более изолированно друг от друга, чем это может быть достигнуто с помощью распространенных сейчас машинно-ориентированных языков. Логический программист может изобретать разнообразные описания интересующей его задачи и рассматривать, как каждое из них реагирует на механизмы управления, предлагаемые имеющимися в его распоряжении интерпретатором. В конечном итоге должны выбираться та логика и то управление, которые дают максимальную ясность программы при условии ее исполнения с приемлемой эффективностью.

Когда выбор управления довольно ограничен, как это имеет место в стандартных интерпретаторах Пролога, программист иногда бывает вынужден ради эффективности составлять

утверждения с весьма сложной логической структурой. И тем не менее такие утверждения все еще можно анализировать (например, для оценки корректности того, что в них говорится о решаемой задаче), совершенно не учитывая их влияние на поведение программы в период исполнения. Именно эта возможность анализировать по отдельности *декларативные* и *операционные* свойства утверждений придает логике как языку программирования ее отличительные качества.

В данной главе иллюстрируются разнообразные стили логического построения программ и рассматриваются их практические достоинства и недостатки. Излагаемые здесь идеи и суждения вытекают из эмпирических исследований, предпринятых в тот период, когда происходили многочисленные нововведения в технологии реализации, и поэтому не следует считать, что они всегда и при всех обстоятельствах справедливы.

Для того чтобы сделать обсуждения примеров достаточно краткими, принимается ряд допущений относительно интерпретатора, для которого разрабатываются программы. Во-первых, предполагается, если не оговорено противное, что интерпретатор подчиняется стандартной стратегии управления и, стало быть, является (фактически) чистым интерпретатором Пролога. Единственное небольшое отклонение от чистоты связано с использованием встроенных арифметических функций, при помощи которых, как предполагается, термы без переменных, подобные $3 * 2$, вычисляются и заменяются соответствующими сокращениями, в данном случае — константой 6. Во-вторых, без нарушения чистоты предполагается, что у нас имеются некоторые встроенные процедуры, например процедуры для предикатов $=$, \neq и т. д., избавляющие нас от утомительной необходимости добавлять соответствующие комментарии к каждой приводимой программе.

В некоторых примерах порождаемые программами вычисления обсуждаются, но не обязательно иллюстрируются. Все они получаются довольно просто, и читатель лучше усвоит логическое программирование, если построит эти вычисления сам.

III. 1. Логика и управление

Поведение, получающееся в результате исполнения логической программы, образует *алгоритм*. Алгоритм можно не без пользы рассматривать как пару: (логика, управление), *логическая компонента* которой — это множество утверждений, составляющих программу, а *управляющая компонента* — это стратегия исполнения. Таким образом, первой компонентой описывается

задача, которую нужно решить, в то время как второй компонент описывается метод ее решения.

Во многих реализациях управляющая компонента в значительной степени фиксируется в интерпретаторе. Программист может оказывать влияние на управление исполнением своей программы путем выбора текстуального упорядочения вызовов и процедур, а также путем применения ряда специальных средств, таких как оператор отсечения, но общий принцип поиска сверху вниз и в глубину имеет, как правило, первостепенную важность.

Использования лишь текстуального упорядочения, вообще говоря, недостаточно для получения из каждой конкретной логической компоненты широкого диапазона пригодных алгоритмов. Одно упорядочение может дать хороший алгоритм, тогда как все остальные могут вообще не обладать никакими практическими достоинствами. Эффективное логическое программирование при современном состоянии этого искусства основывается поэтому главным образом на выборе соответствующей логической компоненты, удовлетворяющей требованиям имеющегося ограниченного управления. Может быть когда-нибудь окажется возможным, основываясь на чьей-то реализации, изобрести наиболее эффективное управление для совершенно произвольной входной программы, возлагая, таким образом, бремя интеллектуальной деятельности на машину, а не на программиста. Такие условия, однако, не возникнут до тех пор, пока не будет сделано значительных продвижений в теории алгоритмов.

Влияние логической структуры программы на ее поведение в период исполнения можно проиллюстрировать, рассмотрев задачу нахождения числа n различных элементов в некотором заданном списке, например в списке $x = (A, B, C, A, D, B, C, E, A)$. Один простой алгоритм ее решения заключается в том, чтобы сначала отфильтровать все дубликаты из x , получая в результате список $y = (A, B, C, D, E)$, а затем вычислить длину ($n=5$) этого списка. Следующая программа выражает логику, лежащую в основе описанного алгоритма.

Программа 1

```
? подсчитать(A.B.C.A.D.B.C.E.A.NIL,n)
    подсчитать(x,n) если фильтр(x,y), длина*(y,0,n)
    фильтр(NIL,NIL)
    фильтр(u.x,u.y) если удалить(u,u.x,x'), фильтр(x',y)
    удалить(u,NIL,NIL)
    удалить(u,u.x,z) если удалить (u,x,z)
    удалить(u,v.x,v.z) если u ≠ v, удалить(u,x,z)
    длина*(NIL,m,m)
    длина*(u.y,i,m) если длина*(y,i + 1,m)
```

Различные предикаты, встречающиеся в этой программе, читаются следующим образом:

подсчитать (x, n)	в список x входят n различных элементов;
длина* (y, i, m)	длина списка y равна $m - i$;
удалить (u, x, z)	список z получается в результате удаления всех вхождений u из списка x ; относительный порядок оставшихся элементов при этом сохраняется;
фильтр (x, y)	список y получается в результате удаления всех дубликатов из списка x ; относительный порядок оставшихся элементов при этом сохраняется.

Прежде чем двигаться дальше, читателю следует удовлетвориться декларативным содержанием приведенных процедур; их операционные свойства станут тогда более очевидными.

При исполнении этой программы весь подсчет числа различных элементов откладывается до тех пор, пока не будет завершена операция фильтрации, которая удаляет все дубликаты из входного списка. Задания, связанные с фильтрацией и подсчетом, выполняются последовательно благодаря стандартной стратегии управления, согласно которой вызов процедуры **длина *** остается латентным до тех пор, пока не будет решен вызов процедуры **фильтр**. Результат исполнения программы можно резюмировать следующим образом

<u>Списки, подлежащие фильтрации</u>	<u>Результаты подсчета</u>	
($A, B, C, A, D, B, C, E, A$)	нет	элементы списка постепенно отфильтровываются, и в конечном итоге остается список $y = (A, B, C, D, E)$,
(A, B, C, D, B, C, E)	нет	
(A, B, C, D, C, E)	нет	
(A, B, C, D, E)	нет	

<u>Списки, подлежащие подсчету</u>	<u>Результаты подсчета</u>	
(A, B, C, D, E)	0	затем вычисляется длина списка y путем учета и удаления одного за другим его элементов до тех пор, пока не останется только пустой список;
(B, C, D, E)	1	
(C, D, E)	2	
(D, E)	3	
(E)	4	
<i>NIL</i>	5	окончательный результат подсчета — $n = 5$

Сравним теперь полученное поведение с поведением второй программы, в которой предикат подсчитать $^*(x, i, m)$ означает, что число различных элементов в списке x равно $m - i$.

Программа 2

```
? подсчитать(A.B.C.A.D.B.C.E.A.NIL,n)
подсчитать(x,n) если подсчитать*(x,0,n)
подсчитать*(NIL,m,m)
подсчитать*(u.x,i,m) если удалить(u,u.x,z),
подсчитать*(z,i+1,m)
и те же самые процедуры удалить, что и прежде.
```

При исполнении этой программы осуществляется *чредование* процессов подсчета различных элементов и вычеркивания дубликатов. На каждом шаге итерации, который получается посредством вызова второй процедуры подсчитать *, в списке ищется некоторый новый элемент, отличный от всех предыдущих, удаляются все его вхождения в список, и, наконец, он учитывается путем прибавления 1 к счетчику. Все это можно резюмировать следующим образом.

Списки, в которых подсчет и удаление элементов производятся одновременно	Результаты подсчета
(A,B,C,A,D,B,C,E,A)	0
(B,C, D,B,C,E)	1
(C, D, C,E)	2
(D, E)	3
(E)	4
NIL	5

Например, во 2-й строке этого резюме показывается состояние данных в тот момент, когда целевым утверждением становится

? подсчитать*(B.C.D.B.C.E.NIL,1,n)

Элемент A был удален, и число удаленных элементов (*I*) стоит на второй позиции среди аргументов вызова подсчитать *.

Вторая программа является более краткой, чем первая, однако ее общее действие, возможно, менее понятно. Она оказывается к тому же более эффективной, поскольку в ней не требуется строить промежуточный список $y = (A, B, C, D, E)$, состоящий из всех различных элементов. Поведения двух полученных алгоритмов (программа 1, стандартный Пролог) и (программа 2, стандартный Пролог) совершенно отличаются друг от друга несмотря на то, что они обладают общей управляющей компонентой. Это происходит исключительно потому, что в них присутствуют разные логические компоненты; мы говорим, что эти компоненты имеют различное «прагматическое содержание»

относительно стандартного Пролога. Для исполнения этих двух программ помимо стандартной могут быть предложены и другие стратегии. Например, система IC-Пролог в Имперском колледже способна принимать на входе программу 1 вместе с некоторыми небольшими управляющими аннотациями и исполнять ее при помощи стратегии *сопрограмм*. Эта стратегия может попеременно активировать вызовы процедур *фильтр* и *длина**, чередуя таким образом процессы фильтрации и подсчета. Получаемый в результате алгоритм (программа 1, IC-Пролог + аннотации) фактически совпадает с алгоритмом (программа 2, стандартный Пролог).

III.2. Итерация и рекурсия

Для решения многих видов вычислительных задач требуется неоднократно повторять одни и те же алгоритмические процессы. Такие процессы всегда могут быть описаны с помощью рекурсивных или итеративных процедур. Итерация обычно бывает более эффективной, чем рекурсия, поскольку для завершения шага итерации — в отличие от шага рекурсии — не требуется ожидать результатов выполнения последующих шагов. Использование итерации, следовательно, позволяет избежать расходов, связанных с организацией в период исполнения стека латентных вызовов, что невозможно при употреблении рекурсии.

Для того чтобы сравнить итеративный и рекурсивный стили программирования, допустим, что нам требуется написать программу, строящую по некоторому заданному списку x , такому как (A, B, C, D) , обратный список y , который в нашем случае имел бы вид (D, C, B, A) . В приводимой ниже программе 3 эта цель достигается при помощи предиката *обратить* (x, y), означающего, что список y является обратным по отношению к списку x . В ней используется еще один предикат *склеить* ($z1, z2, z$), который означает, что список z получается в результате добавления списка $z2$ в конец списка $z1$. Общая идея программы заключается в том, что из непустого списка x вида $u.x'$ можно получить обратный, строя сначала список y' , обратный по отношению к x' , а затем добавляя $u.NIL$ в конец y' и получая тем самым искомый список y .

Программа 3

```
? обратить(A.B.C.D.NIL,y)
обратить(NIL,NIL)
обратить(u.x',y) если обратить(x',y')
    склеить(y',u.NIL,y)
```

а также процедуры *склеить*, позволяющие склеивать любые два конкретные списки (см., например, программу 11 из разд. III.4.2)

В результате неоднократного вызова рекурсивной процедуры **обратить** по ходу исполнения программы в целевом утверждении образуется стек латентных вызовов склеить, который в конце концов достигнет следующего состояния

```
? обратить(NIL,y4), склеить(y4,D.NIL,y3),
 склеить(y3,C.NIL,y2),
 склеить(y2,B.NIL,y1),
 склеить(y1,A.NIL,y)
```

В этот момент на вызов будет отвечать базисная процедура для предиката **обратить**, и обращение к ней даст присваивание $y := NIL$. Затем по очереди будут решены вызовы склеить, которые, как и ожидалось, дадут выходное присваивание $y := D.C.B.A.NIL$.

Задачу обращения списка можно решить также итеративно, используя более компактную, хотя, возможно, и менее понятную программу, которая приводится ниже.

Программа 4

```
? обратить(A.B.C.D.NIL,y)
 обратить(x,y) если обратить*(NIL,x,y)
 обратить*(y,NIL,y)
 обратить*(x1,u.x2,y) если обратить*(u.x1,x2,y)
```

Здесь предикат **обратить** * ($z1, z2, y$) означает, что список y является обратным по отношению к списку, получаемому в результате добавления $z2$ в конец обращенного списка $z1$. Исполнение этой программы эффективным образом порождает следующее итеративное вычисление:

```
? обратить ( A.B.C.D.NIL,y)
? обратить*(NIL,A.B.C.D.NIL,y)
? обратить*(A.NIL,B.C.D.NIL,y)
? обратить*(B.A.NIL,C.D.NIL,y)
? обратить*(C.B.A.NIL,D.NIL,y)
? обратить*(D.C.B.A.NIL,NIL,y)
? ответ  $y := D.C.B.A.NIL$ 
```

Здесь на каждом шаге итерации из входного списка эффективным образом удаляется некоторый новый элемент, который переносится в начало еще одного списка, где накапливаются уже удаленные элементы. Этот последний список постепенно строится на первой позиции среди аргументов вызовов процедуры **обратить** *, и когда его построение будет закончено, в результате обращения к базисной процедуре **обратить** * он будет присвоен в качестве значения переменной y .

Программа 4 иллюстрирует типичные особенности итеративных логических программ, а именно использование в них специальных параметров для помещения информации, которая в эквивалентной рекурсивной программе была бы закодирована в стеке латентных вызовов. Например, после двух шагов рекурсии в исполнении программы 3 целевым утверждением будет

$$\begin{aligned} ? \text{ обратить}(C.D.NIL, y2), \text{ склеить}(y2, B.NIL, y1), \\ \text{ склеить}(y1, A.NIL, y) \end{aligned}$$

Окончательное решение y кодируется в нем двумя латентными вызовами склеить как результат выполнения двух операций склеивания: сначала $B.NIL$ добавляется к списку $y2$, обратному по отношению к $C.D.NIL$, а затем в конец полученного списка добавляется $A.NIL$. В противоположность этому при исполнении программы 4 по существу та же самая информация кодируется в целевом утверждении

$$? \text{ обратить}^*(B.A.NIL, C.D.NIL, y)$$

с той лишь разницей, что списки $A.NIL$ и $B.NIL$ явно в него не входят, а содержится в нем уже результат $B.A.NIL$ их склеивания, который в свою очередь будет добавлен к списку, обратному по отношению к $C.D.NIL$. (В основе этой тактики лежит ассоциативность операции склеивания: для формального доказательства того, что две рассматриваемые программы вычисляют одно и то же решение для переменной y , необходимо сделать предположение о справедливости данного свойства.)

Подведем итоги нашего обсуждения. В программе 3 используются предикаты обратить и склеить, параметры которых имеют концептуально простые взаимоотношения друг с другом, в то время как в программе 4 употребляется предикат обратить*, чьи три параметра связаны более сложным образом. Более высокая эффективность программы 4 достигается, следовательно, за счет потери логической ясности.

III.3. Поведение сверху вниз и снизу вверх

В типичном последовательном алгоритме на начальном шаге обрабатываются некоторые входные данные, а каждый последующий шаг оперирует с результатами, вычисленными на предыдущих. Заключительный шаг алгоритма выдает желаемые выходные данные. Эта простая парадигма называется «инкрементным» (или пошаговым) решением задач. Данный термин

означает, что решение получается не в результате единственного атомарного вычислительного события, а, напротив, к нему следует постепенно приближаться посредством нескольких взаимозависимых событий. На примере формулировок пошаговых алгоритмов логический программист имеет возможность сравнивать стили программирования *сверху вниз* и *снизу вверх*.

III.3.1. Вычисления факториалов

Два этих стиля программирования можно сравнить, используя задачу вычисления $z = 3!$. Рассмотрим сначала следующую рекурсивную программу

Программа 5

```
?  факториал(3, z)
F1 :  факториал(0, 1)
F2 :  факториал(x, y) если x > 0, факториал (x - 1, y'), y = x * y'
```

Мы уже видели в гл. II, что стандартное исполнение этой программы демонстрирует типичные черты решения задач методом сверху вниз: цель — вычисление значения $3!$ — сводится к подцелям, заключающимся в вычислении значений $2!$ и $3 * 2!$, которые в свою очередь сводятся к дальнейшим подцелям. На каждом шаге этого процесса новые подцели выводятся исходя из стремления решить существующую подцель, так что в целом данное исполнение является целенаправленным. Тот факт, что $0! = 1$, выражаемый процедурой F1, не будет использоваться до тех пор, пока в ходе исполнения программы не будет порожден вызов, специально требующий вычисления значения $0!$.

В противоположность этому программист, пользующийся каким-либо традиционным языком, предпочел бы, вероятно, написать программу вычисления факториала, поведение которой было бы в точности обратным по отношению к обсуждавшемуся выше: каждое полученное значение факториала сразу бы использовалось для вычисления значения очередного, большего факториала, как это делается, например, в следующей программе

```
begin x := 0; y := 1;
  while x ≤ 2 do begin x := x + 1;
    y := x * y
  end
end
```

Эта программа породила бы вычисление методом снизу вверх, в котором значение $1!$ вычислялось бы исходя из значения $0!$,

затем значение 2! исходя из 1! и т. д. Такой метод является итеративным, и он более эффективен, чем рекурсивный метод сверху вниз. Как же тогда логический программист может получить подобный алгоритм?

Одна из возможностей — исполнять программу 5 при помощи нестандартной стратегии управления снизу вверх. Говоря кратко, в этой стратегии используется то обстоятельство, что факт

F3 : факториал(1,1)

логически следует из процедур F1 и F2 и выводим из них посредством шага вывода, называемого резолюцией снизу вверх. Следующий факт

F4 : факториал(2,2)

точно так же выводим из F2 и F3. Успешное вычисление, порождаемое таким способом исполнения, состоит из ряда фактов, за которым следует \square , причем заключительное противоречие возникает тогда, когда какой-либо выведенный факт непосредственно решает исходное целевое утверждение. В нашем примере получилось бы следующее вычисление

F1 :	факториал(0,1)	(дано)
F3 :	факториал(1,1)	(выводится из F1 и F2)
F4 :	факториал(2,2)	(выводится из F3 и F2)
F5 :	факториал(3,6)	(выводится из F4 и F2)
\square		(выводится из F5 и целевого утверждения ?факториал(3,z))

Очевидно, что это поведение точно такое же, как и поведение приведенной выше алголоподобной программы; оно является очень эффективным.

К сожалению, реализовать стратегии управления снизу вверх оказывается гораздо труднее, чем стратегии управления сверху вниз, поскольку для их эффективной работы требуется использовать какие-либо дополнительные средства, ограничивающие множество порождаемых фактов, так чтобы выводились только те из них, которые способствуют решению конкретного целевого утверждения программы: шаг вывода снизу вверх не обладает внутренней целеправленностью, и его неограниченное применение обычно приводит к экспоненциальному росту числа выводимых фактов (в примере с вычислением факториала этого, однако, не происходит). Очень трудно, оказывается, предложить такие средства, обладающие сколько-нибудь полезной степенью общности, и именно поэтому в стандартных реализациях употребляется только метод сверху вниз. В этой ситуации логическому программисту, желающему добиться поведения в соответствии с методом снизу вверх, следует прибегнуть к про-

грамме специального вида, стиль которой мы назовем стилем «квази-снизу вверх»: хотя она и исполняется с помощью стандартной стратегии сверху вниз, ее поведение моделирует исполнение методом снизу вверх. Этот стиль мы проиллюстрируем приводимой ниже программой 6.

Программа 6

```
? факториал(3, z)
  факториал(x, y) если факториал*(0, 1, x, y)
    факториал*(x, y, x, y)
      факториал*(u, v, x, y) если u < x, факториал*(u+1, (u+1)*v, x, y)
```

Здесь предикат **факториал***(*u, v, x, y*) выражает отношение

если $u! = v$ то $x! = y$

Стандартное исполнение этой программы дает очень эффективную итерацию

```
? факториал(3, z)
? факториал*(0, 1, 3, z)
.
.
.
? факториал*(1, 1, 3, z)
.
.
.
? факториал*(2, 2, 3, z)
.
.
.
? факториал*(3, 6, 3, z)
?     ответ z := 6
```

Заметим, в частности, что первые два аргумента в последовательных вызовах **факториал *** содержат в точности ту же самую информацию, которая содержится в фактах, образующих вычисление снизу вверх. Именно по этой причине мы и говорим, что новый стиль моделирует поведение, соответствующее методу снизу вверх.

III.3.2. Нахождение путей в графах

Еще один поучительный пример дает рассмотренная Ковалевским задача нахождения путей в графах. Допустим, что мы хотим найти пути из вершины *A* в вершину *E* в ориентированном

графе, изображенном на рис. III. 1. Для решения этой задачи можно предложить следующую простую программу.

Программа 7

```
? идти( $E$ )
идти( $A$ )
идти( $y$ ) если дуга( $x, y$ ), идти( $x$ )
дуга( $A, B$ ) дуга( $B, C$ ) дуга( $D, C$ )
дуга( $A, C$ ) дуга( $B, D$ ) дуга( $C, E$ )
```

Здесь предикат **идти** (x) означает, что процесс поиска может «идти» в вершину x , а предикат **дуга** (x, y) означает, что в графе

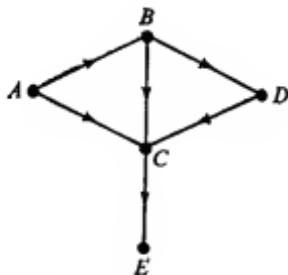


Рис. III. 1. Ориентированный граф.

имеется дуга, направленная из вершины x в вершину y . Стандартное выполнение программы 7 дает различные успешные вычисления, каждое из которых соответствует некоторому пути из A в E , например:

```
? идти( $E$ )
? дуга( $x, E$ ), идти( $x$ )
? идти( $C$ )
? дуга( $x, C$ ), идти( $x$ )
? идти( $A$ )
? ответ "да"
```

Обратим внимание, что этот путь (A, C, E) исследуется в обратном направлении — от вершины E к вершине A — в типичном для метода сверху вниз стиле, причем исходный факт **идти** (A) используется только в самом конце вычисления.

Напротив, в алгоритме снизу вверх факт **идти** (A) использовался бы сразу, и затем осуществлялось бы движение вперед по указанному пути: следующей была бы найдена вершина C ,

а потом и E . Этого эффекта можно достичь с помощью программы, написанной в стиле «квази-снизу вверх».

Программа 8

```
? идти*(A,E)
идти*(x,x)
идти*(x,z) если дуга(x,y), идти*(y,z)
и те же самые факты дуга, определяющие граф
```

Предикат $\text{idti}^*(x, z)$ означает, что

если процесс поиска может идти в вершину x , то
он может идти и в вершину z

В результате стандартного исполнения этой программы получились бы вычисления, подобные тому, которое приводится ниже:

```
? идти*(A,E)
? дуга(A,y), идти*(y,E)
? идти*(C,E)
? дуга(C,y), идти*(y,E)
? идти*(E,E)
? ответ "да"
```

Очевидно, что это вычисление моделирует поведение, которое было бы получено в результате исполнения программы 7 с помощью стратегии снизу вверх.

III.3.3. Задача нахождения собственных значений и собственных векторов матрицы

Задачи вычисления факториала и нахождения путей в графе с приемлемой эффективностью можно решить как с помощью метода сверху вниз, так и с помощью метода снизу вверх. Имеются, однако, и другие задачи, для решения которых методы сверху вниз оказываются крайне неэффективными, в результате чего существенным инструментом становится стиль «квази-снизу вверх». Хороший пример, ориентированный в некоторой степени на читателя с большими математическими склонностями, дает задача нахождения собственных значений и собственных векторов: по данной матрице M размером $n \times n$ вычислить скаляры (собственные значения) e_i и неупорядоченные векторы (собственные векторы) X_i , удовлетворяющие равенству

$$M \cdot X_i = e_i \cdot X_i$$

Хотя эта задача в общем случае имеет n решений $(e_1, X_1), \dots, (e_n, X_n)$, часто требуется найти только «доминантный» соб-

ствениий вектор, т. е. тот собственный вектор, который соответствует наибольшему собственному значению.

Так, например, задача нахождения собственных значений и собственных векторов матрицы

$$M = \begin{bmatrix} 3 & 2 \\ 3 & 4 \end{bmatrix}$$

размером 2×2 имеет два решения:

$$e_1 = 6, \quad X_1 = C \cdot \begin{bmatrix} 1 \\ 1.5 \end{bmatrix}$$

$$e_2 = 1, \quad X_2 = D \cdot \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

где C и D — произвольные константы. Доминантным собственным вектором здесь является X_1 , поскольку $e_1 = 6$ — наибольшее собственное значение.

В случае когда структура матрицы M такова, что все ее собственные векторы линейно независимы, имеется простой алгоритм определения доминантного собственного вектора. Начиная с произвольным образом выбранного начального приближения V , алгоритм строит последовательность векторов $M \cdot V, M^2 \cdot V, M^3 \cdot V$ и т. д., которая обязательно сходится к доминантному собственному вектору. Алгоритм заканчивает работу, когда один из построенных векторов $M^k \cdot V$ удовлетворяет некоторому условию точности приближения. Приводимая ниже программа 9 является рекурсивной формулировкой (методом сверху вниз) этого алгоритма.

Программа 9

```
? прибл(в),точи(М,в)
прибл(1.1.NIL)
прибл(в) если прибл(в'),умнож(М,в',в)
а также процедуры для предикатов точи и умнож
```

Здесь предикат **прибл**(v) означает, что вектор v является очередным приближением; предикат **точи**(M, v) означает, что v достаточно точно приближает доминантный собственный вектор матрицы M , а предикат **умнож**(M, v', v) выражает равенство $v = M \cdot v'$. Детали проверки на точность приближения и операции умножения матриц сейчас для нас несущественны — мы будем просто предполагать, что для выполнения этих заданий уже имеются соответствующие процедуры. В качестве начального

приближения выбран вектор

$$V = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

который в первой процедуре представляется термом $1.1.NIL$. Мы предполагаем также, что M — это матрица из рассмотренного выше примера, и что ее элементы известны процедурам прибл и умнож.

В ходе исполнения программы начальный вектор V подвергается проверке на точность приближения:

$$\begin{aligned} G1 : & ? \text{ прибл}(v), \text{точн}(M, v) \\ G2 : & ? \quad \text{точн}(M, 1.1.NIL) \end{aligned}$$

Если предположить, что V не удовлетворяет требуемой точности, то исполнение возвращается назад к целевому утверждению $G1$ и затем продолжается следующим образом:

$$\begin{aligned} G2' : & ? \text{ прибл}(v'), \text{умнож}(M, v', v), \quad \text{точн}(M, v) \\ G3 : & ? \quad \text{умнож}(M, 1.1.NIL, v), \text{точн}(M, v) \\ G4 : & ? \quad \text{точн}(M, 5.7.NIL) \end{aligned}$$

Тем самым было выполнено умножение матрицы M на вектор V и получено следующее приближение $M.V = 5.7.NIL$. Если оно снова не удовлетворяет требуемой точности, то исполнение программы опять возвратится назад, и при этом вычисление вектора $5.7.NIL$ будет забыто. Это происходит потому, что всякий раз, когда осуществляется процесс возврата из тупиковой вершины дерева поиска, отбрасывается вся информация о том, что уже было сделано (включая и найденные присваивания), начиная с ближайшей точки ветвления. После возврата из данной тупиковой вершины исполнение программы будет продолжено так:

$$\begin{aligned} G3' : & ? \text{ прибл}(v''), \text{умнож}(M, v'', v'), \text{умнож}(M, v', v), \\ & \quad \text{точн}(M, v) \\ & \quad \vdots \\ G6 : & ? \text{ точн}(M, 29.43.NIL) \end{aligned}$$

При переходе от $G3'$ к $G6$ во второй раз выполняется умножение матрицы M на вектор $1.1.NIL$, в результате чего получается вектор $5.7.NIL$, на который еще раз умножается M , и получается следующее приближение $M^2.V = 29.43.NIL$. Эти два умножения также будут забыты и впоследствии выполнены заново, если и последнее приближение не удовлетворяет условиям точности. Очевидно, это очень неэффективный алгоритм, несмотря на то что логика программы представляется таким прямым описанием задачи. Тем не менее исполнение программы даст в конце кон-

цов для целевой переменной v искомое решение¹⁾. Мы уже можем наблюдать, что вычисленные до сих пор приближения

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 5 \\ 7 \end{bmatrix} \quad \begin{bmatrix} 29 \\ 43 \end{bmatrix}$$

или, иначе,

$$\begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad 5 \cdot \begin{bmatrix} 1 \\ 1.4 \end{bmatrix} \quad 29 \cdot \begin{bmatrix} 1 \\ 1.483 \end{bmatrix}$$

сходятся к доминантному собственному вектору

$$C \cdot \begin{bmatrix} 1 \\ 1.5 \end{bmatrix}$$

Поставленную задачу можно решить гораздо более эффективно, пользуясь стилем «квазн-снизу вверх» аналогично тому, как это сделано в формулировке предиката идти* в программе 8.

Программа 10

```
? вывести(v, 1.1.NIL), точи(M, v),/
вывести(v, v)
вывести(v, z) если умнож(M, z, z'), вывести(v, z')
и, как и прежде, процедуры для предикатов точи и
умнож
```

Здесь предикат **вывести**(v, z) означает, что приближение v выводится из приближения z посредством многократного (в частности, нулькратного) умножения на него матрицы M . Стандартное выполнение этой программы дает теперь превосходный итеративный алгоритм:

```
? вывести(v, 1.1.NIL), точи(M, v),/
.
.
.
? вывести(v, 5.7.NIL), точи(M.v),/
.
.
.
? вывести(v, 29.43.NIL), точи(M.v),/
.
.
.
и т. д.
```

¹⁾ После этого, однако, работа не закончится, поскольку имеется возможность (с помощью второй процедуры **прнбл**) находить все более точные приближения. — Прим. перев.

Вслед за каждым из приведенных выше целевых утверждений вызывается первая процедура `вывести`, в результате чего последнее из полученных приближений подвергается проверке на точность; если оно не удовлетворяет требуемой точности, то вызывается другая процедура `вывести`, действие которой состоит в выполнении однократного умножения матрицы M на вектор и получении, таким образом, следующего приближения. Отметим, что в целевом утверждении используется оператор отсечения: он служит для того, чтобы прекратить исполнение программы, как только будет вычислено первое достаточно точное решение, предотвращая тем самым дальнейшие ненужные уточнения.

Фундаментальное различие между формулировками методом сверху вниз задачи вычисления факториала (программа 5) и задачи нахождения собственного вектора (программа 9) заключается в том, что одна из них является детерминированной, а другая — недетерминированной. Во второй программе каждый активированный вызов `прибл` может обратиться к любой из двух процедур `прибл`, что приводит к очень разветвленному дереву поиска, многие тупиковые ветви которого содержат многократно дублируемые цепочки (дорогостоящих) умножений матрицы на вектор. В первом же случае исполнение программы оказывается детерминированным вплоть до момента вычисления решения.

III.4. Детерминизм и недетерминизм

Способность логических интерпретаторов осуществлять целенаправленный поиск позволяет во многом избавиться от утомительной работы, обычно ожидаемой при программировании для ЭВМ. Указанное свойство интерпретаторов спасает программиста от необходимости слишком уж подробно вникать в детали исполнения своих программ. С особенной очевидностью это преимущество логического программирования проявляется в программах, написанных в недетерминистском стиле: программист знает, что интерпретатор исследует различные пути, не требуя при этом от него какого-либо руководства посредством управляющих директив. Более того, для одних и тех же задач недетерминированные программы оказываются часто более простыми и элегантными, чем их детерминированные варианты. Все это станет понятным из рассмотрения следующих примеров.

III.4.1. Поиск в списке

Допустим, что в некотором списке L требуется найти позицию (если, конечно, она имеется) с наименьшим номером, на которой стоит элемент, удовлетворяющий определенному свой-

ству. L может быть, к примеру, списком чисел $(4, 7, -3, 0, -2)$, а разыскиваемый элемент может удовлетворять, например, свойству «быть отрицательным числом». Простым представлением списка L являлось бы тогда множество фактов:

$$\begin{aligned}\exists(4, 1, L) \\ \exists(7, 2, L) \\ \exists(-3, 3, L) \\ \exists(0, 4, L) \\ \exists(-2, 5, L)\end{aligned}$$

где предикат $\exists(u, i, x)$ означает, что элемент u занимает в списке x позицию с номером i . Формулируя только целевое утверждение

$$? \exists(u, k, L), u < 0$$

мы получаем чрезвычайно простую недетерминированную программу, стандартное исполнение которой выдаст два решения $k := 3$ и $k := 5$ в указанном порядке. Программисту известно, что первое из них является решением исходной задачи, поскольку он знает, что интерпретатор будет вызывать факты \exists согласно их текстуальному упорядочению, а оно в свою очередь организовано в соответствии с упорядочением элементов в списке L . От второго решения можно было бы, конечно, избавиться с помощью оператора отсечения, формулируя целевое утверждение

$$? \exists(u, k, L), u < 0, /$$

Во всяком случае заключение о том, что первое из вычисленных решений дает наименьший номер позиции, на которой стоит отрицательный элемент, зависит здесь от управляющей компоненты алгоритма, т. е. от того предположения, что исполнение программы управляет стандартной стратегией; это заключение не вытекает из логики программы, и именно поэтому ее логика оказывается столь простой.

Ценность такого стиля программирования зависит частично от того, насколько удобно представлять элементы списка L в соответствующем порядке. Если бы фактов \exists было намного больше и в результате некоторого предварительного процесса подготовки данных их порядок был бы каким-то образом нарушен, то, возможно, оказалось бы предпочтительнее использовать детерминированную программу, которая совершенно не за-

висит от текстуального упорядочения этих фактов. Вот одна подобная программа.

```
? позиция( $k, i, 5, L$ )
позиция( $k, i, n, x$ ) если  $\exists(u, i, x), u < 0, k = i$ 
позиция( $k, i, n, x$ ) если  $\exists(u, i, x), u \geq 0,$ 
позиция( $k, i + 1, n, x$ )
и те же самые факты, представляющие  $L$  и
расположенные в произвольном порядке
```

Здесь предикат **позиция** (k, i, n, x) означает, что k есть наименьший номер позиции отрицательного элемента в подсписке списка x , начинаящегося с позиции i и продолжающегося до позиции n включительно. Данная программа имеет только одно решение $k := 3$, и тот факт, что оно дает наименьший номер позиции отрицательного элемента, логически вытекает теперь из ее процедур. Таким образом, упростив требования к управлению (путем ослабления всех допущений относительно текстуального упорядочения) и тем самым облегчив решение задачи подготовки данных, мы оказались вынужденными составлять логически более сложную программу, чем та, которая была в предыдущем случае. Обе эти формулировки представляют собой фактически один и тот же алгоритм с небольшими различиями, касающимися эффективности.

III.4.2. Обнаружение пика

Написание недетерминированных программ может быть очень простым делом, однако не всегда они могут эффективно исполняться с помощью средств стандартного управления. Хороший пример для иллюстрации этого факта дает задача, в которой требуется определить, является ли последовательность чисел *униодальной* в том смысле, что сначала она возрастает до некоторого единственного максимума, а затем убывает (т. е. имеет единственный пик). Имеет так ведет себя, например, последовательность $L = (1, 3, 4, 8, 6, 5, 2, 0)$, пиком которой оказывается число 8. Наиболее простым способом описания этой задачи на языке логики является, быть может, следующий.

Программа 11

```
? пик(1..4..8..6..5..2..0..NIL)
пик( $x$ ) если склейть( $x_1, x_2, x$ ) вверх( $x_1$ ), вниз( $x_2$ )
вверх(NIL)
вверх( $u..NIL$ )
вверх( $u..v..y$ ) если  $u < v$ , вверх( $v..y$ )
вниз(NIL)
```

вниз(*u.NIL*)

вниз(*u.v.y*) если *u > v*, вниз(*v.y*)

склеить(*NIL,x,x*)

склеить(*v.y₁,y₂,v.y*) если склеить(*y₁,y₂,y*)

Здесь предикат **пик(х)** означает, что в списке *x* имеется пик; предикат **склеить(*x₁,x₂,x*)** означает, что список *x* получается в результате добавления списка *x₂* в конец списка *x₁*; **вверх(*x₁*)** означает, что последовательность *x₁* является возрастающей, а **вниз(*x₂*)** — что последовательность *x₂* убывающая, причем каждая из этих двух последовательностей может быть пустой¹⁾.

Когда вызывается процедура **пик.** она пытается разделить входной список на две части, первая из которых представляет собой возрастающую последовательность, а вторая — убывающую; это разделение осуществляется процедурами **склеить.** Поскольку на порожденные в ходе исполнения программы вызовы **склеить** отвечают обычно обе процедуры с этим именем, алгоритм ведет себя недетерминированно, пытаясь исследовать все возможные способы разделения входной последовательности. Такое поведение имеет два существенных недостатка. Во-первых, вслед за каждым новым разбиением будут полностью проводиться полученные новые последовательности *x₁* и *x₂*, скажем

$$x_1 = (1, 3, 4)$$

$$x_2 = (8, 6, 5, 2, 0)$$

даже если в результате предыдущего разбиения, такого как

$$x'_1 = (1, 3)$$

$$x'_2 = (4, 8, 6, 5, 2, 0)$$

уже было, возможно, установлено правильное упорядочение некоторых частей последовательностей *x₁* и/или *x₂*. Во-вторых, в том случае, когда входная последовательность не удовлетворяет свойству **пик.** программа не распознает этот факт до тех пор, пока не испробует все возможные разбиения. Это наблюдение привлекает внимание к одному важному моменту в методологии логического программирования: оказывается недостаточно рассматривать эффективность только успешных вычислений; необходимо, кроме того, попытаться сделать все, чтобы и в случае отсутствия решений исполнение программы **эффективно завершалось неудачей.**

¹⁾ Эта программа не совсем точно описывает исходную задачу, поскольку ответом на целевое утверждение ?пик(*1.2.2.1.NIL*) будет «ДА» несмотря на то, что в списке *(1, 2, 2, 1)* имеется два максимума. Заметим, что программы 12 и 19, предназначенные для решения той же самой задачи, дадут на указанное целевое утверждение ответ «НЕТ». В качестве упражнения читатель может сам привести программу 11 в соответствие с точным определением пика.— Прим. перев.

Указанные недостатки нельзя исправить с помощью какого-либо прямого использования стратегии управления, и, стало быть, если требуется более хороший алгоритм, то следует изменить саму логику программы. Приводимая ниже программа дает гораздо более эффективный и детерминированный алгоритм, однако она не является таким уж очевидным описанием спецификации задачи.

Программа 12

```
?ник(1.3.4.8.6.5.2.0.NIL)
ник(NIL)
ник(u.NIL)
ник(u.v.y)если u < v, ник(v.y)
ник(u.v.y)если u > v, вниз(v.y)
и те же самые процедуры вниз, что и прежде
```

Разбиение входной последовательности становится теперь неявно составной частью логики процедур ник. В ходе выполнения программы третья процедура ник используется для просмотра последовательности с целью нахождения ее наибольшей возрастающей подпоследовательности, начинающейся с первого элемента списка *L*. Как только обнаруживается убывающая пара элементов, управление переходит к четвертой процедуре, которая определяет, является ли оставшаяся часть последовательности убывающей. При этом происходит очень мало дублирующих друг друга сравнений элементов, и в случае неразрешимости целевого утверждения исполнение программы завершается, как только будет найдена неправильно упорядоченная пара. Заметим, что для этих усовершенствований потребовалось не введение каких-либо новых предикатов, а лишь более искусные утверждения относительно тех, которые уже использовались; на самом деле мы обошлись даже без предикатов вверх и склейте.

III.4.3. Задача о подстроке

Всякую недетерминированную программу можно преобразовать в более детерминированную, так чтобы ее выполнение моделировало выполнение первой программы. Это преобразование не меняет алгоритма, а предназначено скорее для того, чтобы описать в логике новой программы те аспекты алгоритма, которые были реализованы посредством управления исполнением исходной программы.

Мы рассмотрим здесь эту идею на примере задачи о подстроке: по двум заданным строкам символов *x* и *y* определить, является ли *x* подстрокой *y*. Так, например, строка символов *x* = (*A, B, C*) есть подстрока строки *y* = (*A, B, A, B, C, D*). По-

доброго рода задачи возникают в таких приложениях, как редактирование текстов и библиографический поиск данных. Довольно простой алгоритм решения этой задачи заключается в том, чтобы последовательно порождать все суффиксы строки y , каждый раз проверяя, не является ли строка x префиксом текущего суффикса. В данном контексте суффикс y' строки y — это такая подстрока y , которая продолжается до самого ее конца, а строка x есть префикс y' , если она является подструктурой y' , начинающейся с ее первого символа. Указанный алгоритм из строки $y = (A, B, A, B, C, D)$ образует суффиксы

(A, B, A, B, C, D)

$$(B, A, B, C, D)$$

(A,B,C,D).....здесь строка(A,B,C) — префикс и т. д.

н, стало быть, найдет суффикс (A, B, C, D) , префиксом которого является строка $x = (A, B, C)$, получая тем самым решение нашей задачи.

Логику, лежащую в основе этого процесса, довольно точно можно сформулировать с помощью предиката $\text{строка}(x, y)$, означающего, что x есть подстрока y , и предиката $\text{префикс}(x, y)$, означающего, что x есть префикс y . Представляя строки посредством обычных структурированных термов, только что описанный алгоритм можно получить путем стандартного исполнения следующей недетерминированной программы.

Программа 13

строка1 :	строка($A.B.C.NIL$, $A.B.A.B.C.D.NIL$)
строка2 :	строка(x, y) если префикс(x, y)
префикс1 :	префикс(NIL, y)
префикс2 :	префикс($v.x', v.y'$) если префикс(x', y')

Ниже приводится получаемое в результате поведение, описанное в терминах применения проверок префикс к последовательным суффиксам строк (A, B, A, B, C, D) . (Здесь показаны только три первых вычисления, в которых точки и константы NIL для наглядности опущены.)

? строка(*ABC,ABABCD*) выбирается первый суффикс
? префикс(*ABC,ABABCD*) начинается проверка префикса

? префикс(BC , $BABCD$)
? префикс(C , ABC, D)

*выбирается первый суффикс
начинается проверка
префикс*

неудача, поэтому происходит возврат

? строка(*ABC, ABABCD*)
? строка(*ABC, BABCD*)
? префикс(*ABC, BABCD*)
■

выбирается второй суффикс
и начинается проверка префикса
неудача, поэтому
происходит возврат

? строка(*ABC, BABCD*)
? строка(*ABC, ABCD*)
? префикс(*ABC, ABCD*)
? префикс(*BC, BCD*)
? префикс(*C, CD*)
? префикс(*NIL, D.NIL*)
? задача решена, ответ "да"

выбирается третий суффикс
и начинается проверка префикса

Заметим, что *всякий* раз, когда проверка префикса заканчивается неудачей, исполнение программы возвращается назад к самому последнему из активированных вызовов строки. Ранее этот вызов обращался к процедуре строка1 для того, чтобы инициализировать (неудачную) проверку префикса для текущего суффикса. Теперь же он должен быть активирован снова и вместо процедуры строка1 вызвать процедуру строка2, порождая тем самым следующий суффикс. Такое поведение происходит *всякий* раз, когда исполнение программы наталкивается на тупиковую вершину, которых в общем случае довольно много.

Когда интерпретатор осуществляет процесс возврата, он эффективным образом вспоминает, что начальной входной строкой *x* является строка (*A, B, C*), и, кроме того, он определяет, какой вид имеет текущий суффикс. Заметим, что непосредственно из целевого утверждения в тупиковой вершине извлечь эту информацию не удается. Так, например, первое вычисление заканчивается неудачей, когда целевое утверждение есть

? префикс(*C, ABCD*)

и само по себе оно не сообщает ничего полезного ни о строке *x*, ни о текущем суффиксе. Требуемая информация восстанавливается из какого-то другого целевого утверждения, встретившегося ранее в ходе исполнения программы, доступ к которому можно получить теперь при помощи механизма возврата.

Можно составить гораздо более детерминированную программу, в которой при неудачном завершении проверки префикс нет нужды возвращаться назад для того, чтобы извлечь информацию, необходимую для обработки следующего суффикса. Эту информацию программа получает теперь из дополнительных параметров, содержащихся в текущем целевом утверждении, что достигается за счет использования нового предиката строка*(*w, z, x, y'*), который читается как

w есть префикс *z* или *x* есть подстрока *y'*

Данный предикат вводится в предвидении той стадии исполнения программы, когда в ходе выяснения, является ли строка x подстрокой некоторого текущего суффикса $v.y'$, потребуется решать будет ли w префиксом z ; если эта проверка префикс заканчивается неудачно, то следующий суффикс y' находится на четвертой позиции среди аргументов предиката **строка***, а строка x — на третьей. Так, например, первая неудача в рассмотренной выше программе происходит для случая, когда $w = C.NIL$, $z = A.B.C.D.NIL$, $x = A.B.C.NIL$, а $y' = A.B.A.B.C.D.NIL$. Новая программа такова:

Программа 14

```
? строка(A.B.C.NIL,A.B.A.B.C.D.NIL)
строка3 : строка(NIL,y)
строка4 : строка(x,v.y') если строка*(x,v.y',x,y')
строка*1 : строка*(NIL,z,x,y')
строка*2 : строка*(NIL,z,x,y') если строка(x,y')
строка*3 : строка*(v.w,v.z,x,y') если строка*(w,z,x,y')
строка*4 : строка*(u.w,v.z,x,y') если u ≠ v, строка(x,y')
```

Здесь процедура строка3 предназначена для того случая, который в предыдущей программе обрабатывался процедурой префик1, а процедура строка4 инициирует сразу проверку префикса и выбор суффикса, что прежде делалось по отдельности двумя процедурами строка1 и строка2. Процедуры строка*1 и строка*3 являются прямыми аналогами процедур префикс1 и префикс2. В них проверка префикса применяется к первым двум параметрам предиката **строка***, в то время как запись строки x и следующий суффикс y' сохраняются в последних двух параметрах. В процедурах строка*2 и строка*4 представлена суть новой программы, поскольку их логика предназначена для исследования очередного суффикса после завершения проверки префикса для текущего суффикса; в первой процедуре рассматривается успешный, а во второй — неудачный результат этой проверки. В частности, если проверка заканчивается неудачно, то вызывается процедура строка*4, которая вводит строки x и y' в новую фазу вычисления, обрабатывающую следующий суффикс. Ниже приводится часть исполнения новой программы.

```
? строка( ABC,ABABCD)
? строка*(ABC,ABABCD,ABC,BABCD) обрабатывается первый
   суффикс
? строка*( BC, BABCD,ABC,BABCD)
? строка*( C, ABCD,ABC,BABCD)
   проверка префикс заканчивается
   неудачей
? C ≠ A, строка(ABC,BABCD)
```

? строка(*ABC,ABCD*)

? строка*(*ABC,ABCD,ABC,ABCD*)

обрабатывается второй
суффикс

и т. д.

Когда проверка префикса заканчивается неудачей для первого суффикса вследствие того, что целевое утверждение

? строка*(*C,ABCD,ABC,ABCD*)

не может вызвать процедуру строка*3, это утверждение содержит достаточно информации для того, чтобы породить следующий суффикс и инициировать его исследование. Таким образом, механизм возврата при неудачном завершении проверки префикса не потребуется. Алгоритм исполнения первой программы остался тем же самым, но теперь он реализован при помощи уже другого сочетания логики и управления. Эффективность при этом не обязательно повышается; соотношение расходов, связанных с управлением процессом возврата и обработкой более сложных предикатов, будет зависеть от конкретного используемого интерпретатора. Однако показанный здесь вид преобразования программ оказывается важным для других целей. А именно логические представления гораздо более сложных алгоритмов решения задачи о подстроке можно получить из программы 14 более просто, чем из программы 13. Эти алгоритмы в решающей степени зависят от детального логического анализа неудачных завершений проверок префикса, базирующегося на том виде информации, который содержится в вызовах строка* в программе 14.

III.5. Отрицание

Довольно часто программист сталкивается с задачами, наиболее естественные формулировки которых требуют средств для выражения отрицания «не». Например, может возникнуть желание обратиться к базе данных с запросом; верно ли что такой-то элемент в ней *не* содержится? В полной логике первого порядка или даже в дизъюнктах общего вида отрицание можно выразить непосредственно с помощью символа отрицания \neg . Данный символ, однако, отсутствует в более ограничительных предложениях (хориовских дизъюнктах), используемых для записи логических процедур. Этот недостаток можно исправить различными способами. Для иллюстрации некоторых из них мы рассмотрим конкретную программистскую задачу. Имеются список *x* и элемент *i*. Если *i* не входит в *x*, то требуется вставить *i* куда-либо в *x*, в противном случае — вычеркнуть *i* из *x*. В результате порождается новый список *y*. Точные детали операций

вставки и вычеркивания несущественны. Важным является то, что программа должна решить, выполняется условие задачи или нет, и затем произвести соответствующее действие. Здесь обсуждаются четыре различные логические формулировки данной задачи.

ФОРМУЛИРОВКА 1

Пусть предикат *новсписок* (*u*, *x*, *y*), означает, что *y* — новый список, полученный из *x* в соответствии с тем, содержится элемент *u* в *x* или нет. Предикаты *вычеркнуть* (*u*, *x*, *y*) и *вставить* (*u*, *x*, *y*) означают соответственно, что *y* получается в результате вычеркивания *u* из *x* и *y* получается в результате вставки *u* в *x*. Проверку вхождения *u* в *x* можно осуществить с помощью еще двух предикатов *входит* (*u*, *x*) и *не-входит* (*u*, *x*), смысл которых очевиден. Оснащенная этими предикатами и стандартным целевым утверждением, в котором списки представлены обычными термами, первая формулировка такова:

Программа 15

```
? новсписок(D, A.B.C.NIL, y)
    новсписок(u, x, y) если входит(u, x), вычеркнуть(u, x, y)
    новсписок(u, x, y) если не-входит(u, x), вставить(u, x, y)
    входит(u, u.x)
    входит(u, v.x) если входит(u, x)
    не-входит(u, NIL)
    не-входит(u, v.x) если u ≠ v, не-входит(u, x)
    и процедуры для предикатов вставить и вычеркнуть
```

Эта программа достаточно очевидна. Она вычислит некоторое решение вида $y := A.B.C.D.NIL$ в зависимости от того, как именно осуществляется операция вставки. Главный ее недостаток заключается в том, что требуются отдельные процедуры описания отношений *входит* и *не-входит*. Поскольку эти отношения тесно взаимосвязаны, логическое содержание программы должно быть значительно избыточным.

ФОРМУЛИРОВКА 2

Во втором подходе мы обходимся без процедур *не-входит*, интерпретируя *неудачу* при решении вызова *входит* (*u*, *x*) как сигнал к выполнению вставки. Кроме того, в этом подходе требуется использовать оператор отсечения.

Программа 16

```
? новсписок(D, A.B.C.NIL, y)
    новсписок(u, x, y) если входит(u, x), /, вычеркнуть(u, x, y)
    новсписок(u, x, y) если вставить(u, x, y)
    и процедуры для предикатов входит, вставить и
    вычеркнуть
```

Эта программа намного короче и к тому же исполняется намного эффективнее, потому что операция вставки не зависит, как в программе 15, от решения вызова *не-входит*(*D,A,B,C,NIL*); достаточно вместо этого потерпеть неудачу при решении вызова *входит* в первой процедуре, чтобы сразу перейти ко второй. В противном случае, если вызов *входит* будет успешным, активизация оператора / отсечет еще не испытанный выбор второй процедуры (предотвращая, таким образом, вставку), и затем будет выполнена операция вычеркивания.

Хотя программа 16 операционально правильна, тем не менее в ее логике есть серьезный изъян, поскольку вторая процедура *новсписок* не согласуется с имевшимся в виду смыслом предиката *новсписок*: эта процедура утверждает, что список *y* всегда получается из списка *x* путем выполнения операции вставки, тогда как наша спецификация требует, чтобы условием вставки было отсутствие вхождений *u* в *x*. Причина, по которой эта логически неправильная процедура не приводит к неверным результатам исполнения программы, состоит в том, что ее вызов был предусмотрительно ограничен управляющей информацией, содержащейся неявно в упорядочении текста программы, а также использованием оператора отсечения /.

ФОРМУЛИРОВКА 3

Третий подход в принципе подобен первому, однако в нем устраняется необходимость использования отдельных множеств процедур, описывающих взаимно дополнительные отношения, такие как *входит* и *не-входит*. Вместо этого он основывается на явном вычислении ответов *ДА/НЕТ* на запросы о принадлежности в отношениях.

Введем новые предикаты *входит**(*u, x, ДА*) и *входит**(*u, x, НЕТ*), означающие соответственно, что *u* входит в список *x* и *u* не входит в *x*. Еще один предикат *новсписок**(*u, x, y, z*) означает, что *y* получается либо вычеркиванием *u* из *x*, либо вставкой *u* в *x* в соответствии с тем, какое значение — *ДА* или *НЕТ* — принимает переменная *z*. Программу можно написать теперь следующим образом.

Программа 17

```
? новсписок(D, A.B.C.NIL, y)
новсписок(u, x, y) если входит*(u, x, z), новсписок*(u, x, y, z)
новсписок*(u, x, y, DA) если вычеркнуть(u, x, y)
новсписок*(u, x, y, NET) если вставить(u, x, y)
входит*(u, NIL, NET)
входит*(u, u.x, DA)
входит*(u, v.x, z) если u ≠ v, входит*(u, x, z)
и процедуры для предикатов вычеркнуть и вставить
```

В ходе исполнения программы вызов **входит*** вычисляет ответ **ДА** или **НЕТ** для переменной *z* и распределяет его в вызове **новсписок***; в соответствии с этим ответом затем вызывается либо процедура **вычеркивания**, либо процедура **вставки**.

Программа 17 логически правильна, эффективна, достаточно понятна и не зависит от порядка основных процедур. Вероятно, это лучшая альтернатива для использования при наличии интерпретаторов, предлагающих только стандартную стратегию управления. Несмотря на эти преимущества данный метод становится неудовлетворительным, когда он применяется к задачам, содержащим большое число проверок истинности различных отношений, поскольку становится утомительным писать много процедур, аналогичных приведенным выше процедурам для предиката **входит***. Заметим также, что этот метод имеет тенденцию усложнять логическую структуру программы. Например, в программе 17 требуется использовать довольно громоздкое отношение **новсписок*** для того, чтобы обслуживать различные действия, зависящие от ответов **ДА/НЕТ**, вычисляемых процедурой **входит***.

ФОРМУЛИРОВКА 4

Эта последняя формулировка применима к реализациям, в которых допускаются квазиотрицательные вызовы в программах и которые основаны на так называемом «допущении замкнутости мира»: отрицание предиката **P** справедливо, если **P** недоказуем. Такой прием широко известен под названием *отрицание как неудача*. Вот эта программа.

Программа 18

? **новсписок(*D,A,B,C,NIL*)**

новсписок(*u,x,y*) если входит(*u,x*), вычеркинуть(*u,x,y*)

новсписок(*u,x,y*) если ~входит(*u,x*), вставить(*u,x,y*)

и процедуры для предикатов **входит**, **вычеркинуть** и **вставить**

Символ \sim читается как «не», но он умышленно выбран отличным от связки строгого отрицания \neg , поскольку значения их не совсем одинаковы. Операционное действие \sim заключается в следующем. Когда интерпретатор активирует квазиотрицательный вызов вида $\sim P$, он сначала пытается решить **P**. Если эта попытка оканчивается неудачей, то исходный вызов $\sim P$ считается решенным; таким образом, неудача при доказательстве **P** рассматривается как доказательство справедливости $\neg P$. С другой стороны, если вызов **P** решен, то исходный вызов $\sim P$ считается неудачным.

Применение к нашему примеру такое исполнение программы в конце концов, войдя во вторую процедуру **новсписок**, активи-

рут вызов $\sim \text{входит}(D, A, B, C, NIL)$. Интерпретатор поэтому попытается решить вызов $\text{входит}(D, A, B, C, NIL)$, потерпит при этом неудачу и, следовательно, придет к заключению, что D не входит в список (A, B, C) , т. е. будет считать квазиотрицательный вызов $\sim \text{входит}(D, A, B, C, NIL)$ решенным. Стало быть, интерпретатор приступит, как и предполагалось, к выполнению операции вставки.

Положение становится несколько сложнее в ситуации, когда активируется вызов вида $\sim P(x)$, причем переменная x в данный момент является несвязанной. Если последующая попытка решить P будет успешной, и при этом x не заменится никаким термом, то все еще правильно считать вызов $\sim P(x)$ неудачным. Если же, однако, вызов $P(x)$ решается, и происходит присваивание переменной x какого-либо значения, то в этом случае устанавливается лишь справедливость формулы $(\exists x)P(x)$, что не дает нам никакого ответа на вопрос, поставленный квазиотрицательным вызовом $\sim P(x)$, который интерпретируется как $(\exists x)(\neg P(x))$. Интерпретатор не может прийти ни к какому решению относительно этого вызова, и, стало быть, должен прервать исполнение программы и сообщить об операцией ошибке. Именно таким образом ведет себя реализация IC-Пролога.

Обращение с отрицанием в ранних реализациях Пролога также зависело от допущения замкнутости мира. В этих системах позволялось писать процедуры вида

проц : $Q(y)$ если $\text{не}(P(x))$

Вызов $P(x)$ передавался в качестве параметра встроенному множеству процедур

$\begin{cases} \text{не1} : \text{ не}(z) \text{ если } z / , \text{НЕУДАЧА} \\ \text{не2} : \text{ не}(z) \end{cases}$

где z играет роль метапеременной. Снова, если бы переданный вызов $P(x)$ был неудачным (когда он активировался в результате вызова процедуры не1), то вызов $\text{не}(P(x))$ был бы решен посредством последующего вызова процедуры не2. С другой стороны, если бы вызов $P(x)$ решался успешно в процедуре не1, то оператор $/$ отсек бы использование процедуры не2, а псевдо-вызов **НЕУДАЧА** привел бы к неудачному выходу из не1, что и требовалось. Рассматривать состояние связности переменной x в случае успешного решения $P(x)$ здесь не нужно, потому что во всех этих системах процедура проц интерпретировалась как

для всех y $Q(y)$ если $\neg(\exists x)P(x)$

В IC-Прологе, напротив, процедура

$Q(y)$ если $\sim P(x)$

интерпретируется как

для всех $x, y \ Q(y)$ если $\neg P(x)$

что эквивалентно

для всех $y \ Q(y)$ если $(\exists x)(\neg P(x))$

Применение связки \sim для реализации отрицания посредством неудачи не ведет к противоречиям в стандартных программах (на хорновских дизъюнктах) и значительно расширяет ресурсы программиста. Хотя это и не дает той вычислительной силы, которой обладает логическая система со строгим отрицанием, программисты могут тем не менее без ущерба для себя читать $\sim P(x)$, как «не $P(x)$ » при условии, что они помнят об операционном эффекте данного символа. Одна из причин уменьшения силы связки \sim , между прочим, состоит в том, что хотя она позволяет представлять отрицательные запросы, но не предназначается для выражения отрицательных фактов. Вследствие этого даже когда вызов $\sim P(x)$ решается успешно, переменная x остается несвязанной, и поэтому вычисление не дает конкретных ответов для x . Более общая логическая система могла бы, напротив, вычислить ответ $x := 2$, показывающий, что факт $\neg P(2)$ логически следует из процедур программы.

Исполнение программы 18 оказывается неэффективным, поскольку в нем делаются две попытки решить вызов **входит** (*D,A,B,C,NIL*) — по одной в каждой из процедур **новсписок** А именно после того, как попытка решить этот вызов в первой процедуре закончилась неудачей, он испытывается снова в ходе рассмотрения отрицательного вызова из второй процедуры. Такого рода поведение называется «холостым возвратом», и от него можно избавиться, несколько усилив синтаксис программ с помощью конструкции **если — то — иначе**. Так, в IC-Прологе можно заменить две процедуры для предиката **новсписок** одним утверждением

**новсписок(*u,x,y*) если входит(*u,x*) то вычеркнуть(*u,x,y*)
иначе вставить(*u,x,y*)**

что дает желаемый «синтаксический сахар» и позволяет обходиться без явного отрицания. Последняя процедура интерпретируется следующим образом.

Для того чтобы решить вызов **новсписок(*u,x,y*)**
сначала нужно попытаться решить вызов **входит(*u,x*)**;
в случае успеха выполнить операцию вычеркивания;
в противном случае выполнить операцию вставки.

Она выполняется более эффективно, поскольку вызов **входит** обрабатывается только один раз.

III.6. Согласование параметров

Поскольку логический интерпретатор обладает способностью согласовывать параметры, его можно рассматривать как примитивный процессор обработки данных. Хотя он пытается согласовывать лишь данные, представленные в виде термов, этой способности можно найти довольно существенные применения.

В качестве простого примера допустим, что мы хотим определить, совпадают какие-либо два списка или нет. Самый очевидный алгоритм решения этой задачи состоит в простом последовательном сравнении между собой соответствующих элементов списков. Его можно породить при помощи двух процедур

равны(*NIL,NIL*)

равны(*u.x,u.y*) если равны(*x,y*)

Здесь пошаговый процесс, участвующий в решении целевого утверждения вида

?равны(*A.B.C.NIL,A.B.C.NIL*)

явно выделен посредством использования итеративной процедуры. Однако того же самого результата можно достигнуть, используя вместо этого всего лишь одну процедуру

равны(*x,x*)

Теперь для того, чтобы выполнить все необходимые сравнения элементов, программист полагается на имеющийся в интерпретаторе механизм согласования параметров. Помимо того что этот метод более простой, он оказывается, конечно же, и более эффективным, поскольку исполнение программы включает в себя только один вызов процедуры. Более того, эту единственную процедуру можно использовать для сравнения любой пары термов независимо от того, представляют ли они списки, матрицы, деревья или какие-то более сложные структуры данных. Очень немногие из других языков программирования предоставляют пользователю подобное универсальное встроенное средство для сравнения структур данных.

В общем случае в результате согласования параметров получаются различные взаимосвязанные присваивания термов переменным. Эту возможность можно использовать для манипуляции компонентами составной структуры данных на одном шаге. Так, например, процедуру

поддеревья(*t(x,y),x,y*)

можно вызвать для того, чтобы извлечь левое и правое поддеревья *x* и *y* из заданного бинарного дерева *t(x,y)*. Это делается

с помощью целевого утверждения вида

? поддеревья($t(t(A, B), t(C, D)), x, y$)

в результате этого получается ответ $x := t(A, B)$ и $y := t(C, D)$. С другой стороны, целевое утверждение, подобное

? поддеревья($z, t(A, B), t(C, D)$)

строит посредством той же самой процедуры бинарное дерево $z := t(t(A, B), t(C, D))$. Все эти конкретные операции по обработке данных, возникающие в ходе вычислений, выполняются интерпретатором «за кулисами», и поэтому программисту нет надобности описывать их самому.

Разработать программу, в которой значительная часть операций, связанных с обработкой данных, возлагалась бы на единственный шаг согласования параметров, оказывается не всегда возможно, но даже при удачном исходе это может быть не таким уж простым делом, а иногда и вовсе нежелательным. Предположим, что перед нами поставлена задача — присоединить элемент u в конец некоторого списка x и получить новый список y (т. е. выполнить частный случай операции вставки элемента в список). Довольно легко составить соответствующую итеративную программу, скажем такую:

? присоед($D, A.B.C.NIL, y$)
 присоед($u, NIL, u.NIL$)
 присоед($u, v.x, v.y$) если присоед(u, x, y)

Эта программа вычислит ответ $y := A.B.C.D.NIL$. Не так легко, однако, найти другую возможную формулировку

? присоед*($D, A.B.C.w, w, y$)
 присоед*($z, v, z.NIL, v$)

которая также вычисляет ответ $y := A.B.C.D.NIL$, но делает это всего за один шаг. Программа присоед* основывается на неестественной до некоторой степени конструкции — представлении входного списка (A, B, C) как результата вычеркивания какого-то неопределенного «хвостового» фрагмента w из конца списка $A.B.C.w$, и окончательное присваивание переменной y возникает из довольно замысловатой операции согласования параметров. Эта программа, возможно, более эффективна, чем предыдущая, но в то же время ее гораздо труднее понять. Для других задач, таких как изменение порядка элементов произвольного списка на обратный, по-видимому, нет никаких методов решения, в которых требовалось бы только одно обращение к процедуре.

III.7. Переключатели

Многие алгоритмы можно удобно описывать программами, в которых учитывается и корректируется состояние определяемых программистом переключателей, передающих управление тем или иным процедурам. Использование подобных механизмов в логических программах не всегда желательно, поскольку программы в этом случае могут потерять гибкость — они пишутся, возможно, в расчете только на одну управляющую последовательность и поэтому имеют более ограниченную область приложений, чем логически эквивалентные им программы, написанные при меньшем числе допущений относительно их поведения в период исполнения. Тем не менее если программист об этом не заботится, то он может оказать предпочтение точным устройствам задания управления, таким как переключатели, исходя из тех соображений, что они сделают его алгоритмические намерения более заметными.

В качестве примера рассмотрим еще раз задачу, в которой требуется определить, имеет ли список вида $(1, 3, 4, 8, 6, 5, 2, 0)$ пик, т. е. получается ли он в результате добавления убывающей последовательности x_2 к возрастающей последовательности x_1 , где x_1 и x_2 могут, в частности, быть пустыми или единичными списками. Разумным является алгоритм, который на первом этапе просматривает возрастающую часть x_1 до тех пор, пока не будет обнаружена первая убывающая пара элементов (если, конечно, она имеется), а затем переходит ко второму этапу, на котором он подтверждает, что оставшаяся часть списка представляет собой убывающую последовательность. Программа 12 из разд. III.4.2 дает именно такое поведение, используя при этом предикат **пик** (x), выражающий требуемое свойство списка x . Ниже приводится другая программа, логика которой основывается на идее переключателя, имеющего два возможных состояния — ВВЕРХ и ВНИЗ. Этот переключатель передает управление процедурам, соответствующим текущему этапу нашего алгоритма.

Программа 19

```
? пик*(1.3.4.8.6.5.2.0.NIL,z)
  пик*(NIL,z)
  пик*(u,NIL,z)
  пик*(u.v.w,ВВЕРХ) если u < v, пик*(v.w,ВВЕРХ)
  пик*(u.v.w,z(и если > v, пик*(v.w,ВНИЗ))
```

Здесь предикат вида **пик** $*(y, \text{ВВЕРХ})$ означает, что список y состоит из имеющего пик списка y_2 , добавленного в конец возрастающей последовательности y_1 , а предикат **пик***, $(y, \text{ВНИЗ})$ означает, что y — убывающая последовательность.

В ходе исполнения этой программы каждый вызов процедуры **пик***, начиная со второго, содержит константу в качестве второго аргумента, которая играет роль состояния переключателя, определяющего, какую из процедур **пик*** следует применять на текущем этапе алгоритма. Действие переключателя становится очевидным из следующего вычисления, в котором вызовы процедур **>** и **<**, а также точки и константы **NIL** ради краткости опущены.

```
? пик*(13486520, z)
? пик*( 3486520, ВВЕРХ)    первый этап с состоянием
? пик*( 486520, ВВЕРХ)    переключателя ВВЕРХ
? пик*( 86520, ВВЕРХ)
? пик*( 6520, ВНИЗ)      второй этап с состоянием
? пик*( 520, ВНИЗ)       переключателя ВНИЗ
? пик*( 20, ВНИЗ)
? пик*( 0, ВНИЗ)
? задача решена, ответ "да"
```

На первом этапе с переключателем в состоянии ВВЕРХ используется только первая итеративная процедура **пик*** для осуществления процесса просмотра входного списка. Эта процедура эффективным образом «выключается» (больше не вызывается), как только в результате обнаружения первой убывающей пары 8,6 переключатель переходит в состояние ВНИЗ. Затем «включается» вторая итеративная процедура **пик***, которая используется для завершения процесса просмотра.

Заметим, что в этой новой формулировке мы обходимся без необходимости в программе 12 процедуры **вииз** и, следовательно, получаем более короткую программу. Обычно число различных имён процедур в программе можно сократить с помощью более сложных предикатов, которые содержат переменные, играющие роль переключателей. Теоретически каждый алгоритм можно описать логической программой, в которой используется лишь одно имя процедуры (один предикатный символ) при условии, конечно, что имеется достаточно богатый запас компенсирующих символов констант для обозначения состояний переключателей. Кроме того, употребление переключателей подобно тому, как показано выше, — это, вероятно, наиболее точный способ имитации в логических программах использования операторов **GO TO**, предусмотренных, на радость или горе, в традиционных языках программирования.

III. 8. Исторический очерк

Идея представления алгоритмов с помощью раздельного определения их логических и управляющих компонент играет центральную роль в философии логического программирования

и объясняется в статье Ковальского (1979б), озаглавленной

$$\text{алгоритм} = \text{логика} + \text{управление}$$

Эта схема иллюстрируется в упомянутой статье различными комбинациями логики и управления для ряда известных алгоритмов. В статье подчеркивается также, что отделение логики от управления упрощает задачу анализа логических возможностей алгоритма и его эффективности в период исполнения.

Обычно принято считать, что программа является описанием алгоритма, т. е. мы можем написать

$$\begin{aligned}\text{программа} &= \text{описание алгоритма} \\ &= \text{описание (логики} + \text{управления})\end{aligned}$$

В случае логического программирования эту схему можно продолжить еще на один шаг следующим образом:

$$\begin{aligned}\text{программа} &= \text{описание логики} \\ &+ \\ &\quad \text{описание управления}\end{aligned}$$

В большинстве других формальных систем программирования разложить таким образом составление программ не удается, а возможно лишь описывать (логику + управление) как единый составной объект. Так, например, в них, как правило, допускается, чтобы ход исполнения программы регулировался состояниями, в которых находятся переменные, однако этим в свою очередь определяется, какие присваивания будут иметь место и в каком порядке. Логические связи между переменными нельзя в этом случае анализировать и объяснять, не ссылаясь на состояние исполнения программы, что является обстоятельством, пагубно влияющим на различные стороны современной практики программирования. Интерес, проявлявшийся к «структурному программированию» в 70-х годах, касался этой проблемы, однако фундаментальных причин при этом не затрагивалось. Более подробное обсуждение данного вопроса можно найти в гл. VIII.

При наличии логического интерпретатора программист фактически имеет заранее выполнение описание управления, и поэтому его задача главным образом касается составления оставшегося описания логики. Описание управления не зависит от решаемой задачи, тогда как описание логики определяется именно ею. Одно из наиболее важных достижений исследований в области логического программирования как раз и состоит в установлении того факта, что несмотря на простоту и универсальность стратегии управления, принятой в стандартных ин-

терпретаторах, можно тем не менее получить богатый спектр практических алгоритмов. Раньше некоторые критики логики как языка программирования утверждали, что отсутствие точных управляющих директив во входной программе лишило бы программиста средств эффективного управления ее исполнением и что этот дефицит можно было бы компенсировать только за счет введения в интерпретаторы высоконтеллектуальных стратегий решения задач. Такая критика больше не уместна. При условии, что программист правильно понимает поведение имеющегося у него интерпретатора, посредством выбора соответствующей логики входной программы он может, вообще говоря, управлять исполнением своей программы настолько эффективно, насколько пожелает. Разумеется, современные интерпретаторы в качестве приложения к встроенной стратегии действительно предоставляют еще ряд полезных управляющих директив, но это — дополнительные средства, которые программист не обязан использовать.

Успешная защита логического программирования оказалась возможной в результате предпринятого многими исследователями изучения конкретных вычислительных задач. Благодаря их работе постепенно открываются стили логического программирования, наиболее подходящие для алгоритмов, выполняющих тестирование, поиск, итерацию, рекурсию, работающих в режимах сопрограмм и распараллеливания, обеспечивающих экономию пространства, а также реализующих многие другие важные механизмы. Самой первой и обширной коллекцией подобных результатов является отчет Ковальского (1974а) «Логика для решения задач». Среди многих примеров, приведенных в этой работе, имеются оригинальные формулировки задач синтаксического анализа, сортировки и составления планов, которые иллюстрируют взаимодействие логики и управления в алгоритмах сверху вниз и снизу вверх, детерминированных и недетерминированных.

Сравнение итеративного и рекурсивного стилей программирования можно найти в статьях Кларка (1977), а также Кларка и Ковальского (1977). Открытие Ковальским стиля «квази-снизу вверх» оказалось особенно полезным; так, например, заметив, что в «очевидном» определении чисел Фибоначчи при исполнении методом сверху вниз неэффективно повторяются одни и те же операции, он предложил другое возможное определение, моделирующее метод снизу вверх и дающее высокоеффективное поведение. Подобный стиль независимо был обнаружен Хоггером (1976); с его помощью удалось преодолеть неэффективность, возникающую из-за недетерминизма в рекурсивном определении сверху вниз стандартного алгоритма симплекс-метода в линейном программировании; решение задачи нахождения

собственного вектора и собственного значения из разд. III.3.3 основывается на том же самом подходе.

Об обращении в детерминированную форму недетерминированной «квадратичной» программы для задачи о подстроке (программа 13 из разд. III.4.3) впервые сообщалось Хоггером (1978b), который показал затем (1979b), что это преобразование является важным первым шагом на пути логического вывода более сложно получаемых линейных и сублинейных алгоритмов, разработанных соответственно Кнутом, Моррисом и Праттом (1976) и Бойером и Муром (1977).

Кларк первым предложил использовать позиции аргументов в предикатах для помещения явных ответов *ДА/НЕТ*; ему мы обязаны процедурами, которые приводятся в разд. III.5 в третьей формулировке задачи, связанный с отрицанием. Кларк (1978) предпринял также метатеоретический анализ, необходимый для обоснования использования квазиотрицания (\sim). Семантика и реализация квазиотрицания \sim в IC-Прологе описана Кларком и Маккейбом (1980), а общая проблема отрицания и ее взаимодействие с другими свойствами управления в классическом Прологе рассматривается в статье Дала (1980).

Несколько хитроумных способов использования согласования параметров было предложено Терилундом на семинаре по логическому программированию, состоявшемся в Имперском колледже в 1976 г. Некоторые из них основаны на искусном употреблении «списка различий» в представлении структур данных, применявшегося в программе *присоед** из разд. III.6; другие примеры, демонстрирующие полезность этой конструкции, имеются в статье Кларка и Терилуида (1977).

В данной главе оказалось возможным рассмотреть лишь небольшую выборку стилей логического программирования. Гораздо больше примеров можно найти в отчете Терилунда (1975b) «Логическая обработка информации»; книге Ковалевского (1979a) «Логика для решения задач», статье Ковалевского «Алгоритм = логика + управление», из которой взяты приводимые здесь программы 7 и 8 нахождения путей в графе, а также в докторских диссертациях Хоггера (1979a) и Кларка (1979). Программисты, достаточно много работавшие с реализациями Пролога, уже обнаружили, должно быть, целый ряд других приемов, не описанных в литературе.

IV. Структуры данных

В определениях языков программирования структуры данных и процедуры рассматриваются часто как различные виды вычислительных ресурсов. Это предполагаемое различие ясно выражено в известной формуле

$$\begin{aligned} \text{программа} &= \text{алгоритм} + \text{структура данных} \\ \text{т. е.} \\ &= \text{метод обработки} + \text{обрабатываемый объект} \end{aligned}$$

Приведенная формула, кроме того, лежит в основе некоторых методов программирования, ориентирующих в целях достижения гибкости на раздельное определение процедур и связанных с ними конкретных структур данных. Она с особенной очевидностью проявляется в обычных представлениях традиционных языков программирования, подобных Фортрану и Бейсику, и вряд ли полезна для обучения основным вычислительным понятиям начинающих программистов.

Если в указанной схеме мы заменим «алгоритм» на «логику + управление» и выполним несложное преобразование, то мы получим, что

$$\text{программа} = \text{логика} + \text{структура данных} + \text{управление}$$

Теперь в контексте логического программирования сами логические процедуры могут быть использованы как структуры данных и, стало быть, всякое предполагаемое различие между этими объектами пропадает. Термы, разумеется, также служат в качестве структур данных. Таким образом, какие бы виды структур данных ни использовались, их все следует отнести к логической компоненте программы. Более того, в логических программах практически не делается никакого описания управления, поскольку оно находится в компетенции интерпретатора. С учетом всех этих соображений наша схема применительно к логическому программированию сокращается, следовательно, до такой:

$$\text{программа} = \text{логика}$$

что вполне согласуется с нашим употреблением термина «программа» для обозначения множества логических утверждений.

Поскольку основные составные части (термы и процедуры) структур данных уже были рассмотрены, в этой главе не потребуется вводить какие-либо новые объекты. Вместо этого мы сконцентрируем свое внимание на различных способах использования термов и процедур с целью получения конкретных видов обработки данных.

IV.1. Представление и выборка данных

Наиболее простыми элементами данных в логическом программировании являются константы, такие как *10* или *NIL*, которые не имеют внутренней структуры. Мы употребляем их для всевозможных целей: в качестве чисел, элементов строк, переключателей, а иногда и в качестве имен для других элементов данных. Чаще, однако, мы имеем в программах дело со структурированными элементами данных, которые представляют собой некоторым образом организованные наборы простейших элементов. Двумя основными способами представления структурированных данных являются *представление посредством термов* и *представление посредством фактов*, хотя помимо названных существуют также и другие способы представления. Каждый из них обладает различными достоинствами и недостатками по отношению к стилю программирования и простоте реализации.

IV.1.1. Термы и факты

В представлении посредством термов структурированные данные образуются при помощи функциональных символов, позволяющих собирать составляющие их части в группы. Так, например, список (*10, 20, 30*) можно было бы представить термом *10.20.30.NIL*, в котором каждый функтор точки группирует элемент, расположенный слева от него, со стоящим справа хвостовым фрагментом списка. И константы, и структурированные термы можно рассматривать как по существу пассивные объекты, предназначенные для обработки процедурами.

Структурированные термы этого вида покажутся, быть может, довольно чуждыми программистам, знакомым только с такими языками, как Фортран, Бейсик и Алгол, в которых представление структур данных почти полностью основывается на использовании массивов. Различного рода записи структурированных деревьев, определяемые в языках Кобол, PL/1 и Паскаль, более похожи на логические термы, но наибольшее сходство проявляется, по всей видимости, в Лиспе; фактически применение структурированных термов можно свободно рассма-

травить как «лиспоподобный» метод логического программирования.

Главная выгода от использования термов с целью представления данных заключается в той степени компактности и элегантности, которую они могут придавать процедурам, применяемым для их обработки. Классическим примером является склеивание двух списков, в результате чего получается третий список. Если для решения этой задачи используется алголоподобный язык, зависящий от представления данных в виде массивов, то каждый отдельный элемент из двух заданных списков должен быть выбран и откопирован на соответствующей позиции в третьем списке, для чего потребуются обычные атрибуты итеративного порождения индексов, такие, например, как проверка длин списков, определение границ циклов, предотвращение переполнения массива и т. д. В то же время на языке логики мы можем написать просто

$$\begin{aligned} &\text{склейт}(NIL, y, y) \\ &\text{склейт}(u.x, y, u.z) \text{ если склейт}(x, y, z) \end{aligned}$$

и этого будет вполне достаточно для того, чтобы решить вызов вида склейт($A.B.NIL, C.D.NIL, z$) и получить ответ $z := A.B.C.D.NIL$.

С другой стороны, потенциальным недостатком структурированных термов является то, что их использование может привести к значительной неэффективности исполнения программ, вызываемой главным образом вычислительными усилиями, которые требуются для выборки компонент термов, и отчасти нагрузкой, возлагаемой на процесс унификации. Проблему организации эффективного доступа к компонентам можно в значительной степени преодолеть, прибегая к представлению посредством фактов. Так, например, список $(10, 20, 30)$ можно было бы представить следующим образом:

Представление списка 1

$$\begin{aligned} &\exists(10, 1, L) \\ &\exists(20, 2, L) \\ &\exists(30, 3, L) \\ &\text{длина}(L, 3) \end{aligned}$$

Константа L использована здесь для задания имени структуры данных, и последний факт утверждает, что длина списка L равна 3.

Этот стиль приводит иногда к появлению своих собственных проблем, поскольку для составления необходимых процедур выборки данных программист может быть вынужден писать программы на более «низком» (более машинно-ориентированном)

уровне: он может быть вовлечен в рассмотрение границ, индексов, указателей, вопросов, связанных с управлением циклами, экономией памяти и т. п. Короче говоря, если программист желает обеспечить более хороший доступ к данным, он должен ради этого потрудиться. Было бы, однако, ошибочно принять точку зрения, согласно которой термы и факты обладают противоположными качествами по отношению к стилю и эффективности; многое зависит от той конкретной задачи, для которой пишется программа.

IV.1.2. Прямой и косвенный доступ

Предположим, что список L уже каким-то образом представлен, и мы хотим получить ответ на запрос

$$? \text{ } \exists(u, 2, L)$$

в котором спрашивается, что за элемент занимает в этом списке вторую позицию. Определение элемента u осуществляется путем *прямого доступа*, если для этого требуется не более одного шага вычислений; в противном случае оно осуществляется путем *косвенного доступа* (или «вычисляемого» доступа). Обычно мы полагаем, что прямой доступ является более эффективным.

Форма доступа зависит от способа представления данных. Если используется приведенное выше представление списка 1, то на поставленный запрос можно ответить с помощью прямого доступа, поскольку этот вызов \exists непосредственно решается путем обращения ко второму факту \exists , или, другими словами, путем выполнения одного шага вычислений. Это происходит вследствие того, что в нашем представлении данных компоненты списка определяются явно.

Тот же самый список L можно было бы представить иначе с помощью следующего множества утверждений, в которых эти компоненты определяются неявно:

Представление списка 2

$$\begin{aligned} &\exists(u, i, L) \text{ если } \text{длина}(L, n), 1 \leq i, i \leq n, u = i * 10 \\ &\text{длина}(L, 3) \end{aligned}$$

Решение нашего запроса не сводится теперь к непосредственному просмотру ряда пассивных фактов; напротив, это новое представление ведет себя динамическим, процедурным образом, вынуждая с целью выборки второго элемента списка L строить вычисление, состоящее из нескольких шагов. Хотя этот косвенный доступ оказывается более медленным, при правильном сравнении относительной полезности двух приведенных представлений следовало бы также учитывать и другие факторы, такие как объем памяти, необходимой для хранения списка L ;

в случае когда список L большой и имеет регулярную внутреннюю структуру, второй метод является, очевидно, более компактным, чем первый.

Когда употребляются структурированные термы, стиль программирования становится ориентированным на использование процедур косвенной выборки данных, которые шаг за шагом собирают или разбирают термы для того, чтобы обработать составляющие их компоненты. Мы проиллюстрируем это, применив

Представление списка 3

$\text{список}(L, 10.20.30.NIL)$

в котором просто утверждается, что L — это список $(10, 20, 30)$.

Для того чтобы решить запрос $\exists(u, 2, L)$, мы должны теперь предусмотреть некоторые процедуры, обеспечивающие доступ к элементам списка, такие как

$$\begin{aligned} \exists(u, 1, x) &\text{ если } \text{список}(x, u.x') \\ \exists(u, i, x) &\text{ если } \text{список}(x, v.x'), \exists(u, i - 1, x') \end{aligned}$$

В общем случае в результате вызова этих процедур терм, представляющий список L , будет постепенно разбираться до тех пор, пока не будет найдена требуемая компонента.

Множества процедур, представляющие структурированные данные, обладают интересным и иногда полезным свойством: из них можно образовывать другие возможные представления. Так, например, из представления списка 2 логически следует представление списка 1, и первое из них можно было бы снабдить такими инструкциями, используя соответствующие управляющие директивы, которые позволили бы получить на выходе второе представление. В этом контексте представление списка 2 вело бы себя подобно обычному множеству процедур, порождающему выходные данные. Такая способность логических утверждений одновременно выполнять функции как обычных процедур, так и представлений структур данных показывает, что всякое предполагаемое различие между процедурами и данными носит в сущности pragматический характер, и касается оно лишь использования этих ресурсов, а не присущих им атрибутов.

IV.2. Представления посредством структурированных термов

IV.2.1. Некоторые общие типы данных

Все структурированные термы можно рассматривать как деревья. Например, терм вида $f(t_1, t_2, \dots, t_n)$ можно рассматривать как дерево, изображенное на рис. IV.1a. По этой причине

представления посредством термов оказываются часто в высшей степени пригодными для тех структур данных, которые по своей природе относятся к древесному типу данных. Так, дерево, изображенное на рис. IV. 1b, каждая вершина которого снабжена некоторой меткой, прямо представляется термом $t(t(D, B, E), A, t(F, G, G))$, где каждый подтерм вида $t(u, x, v)$ представляет дерево с корневой вершиной, помеченной буквой x , и двумя расположенными непосредственно ниже нее поддеревьями u и v . Таким же образом дерево, изображенное на рис. IV. 1c, прямо представляется термом $f(f(TIP, TIP), TIP)$.

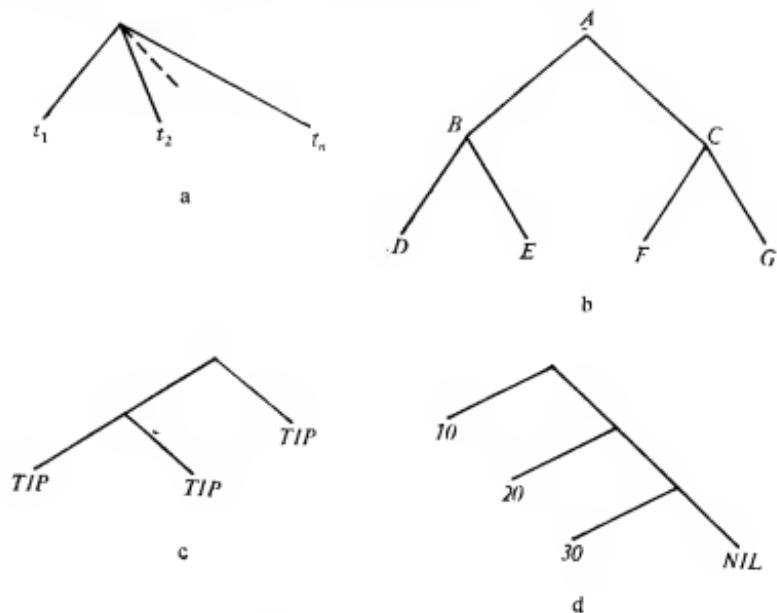


Рис. IV. 1. Древесная структура термов.

В тех случаях, когда мы используем терм, подобный $10.20.30.NIL$, для представления списка $(10, 20, 30)$, этот терм в стандартной префиксной записи приобретает вид $(10, (20, (30, NIL)))$ и, стало быть, имеет древесную структуру, изображенную на рис. IV. 1d. Заметим, в частности, что чем больше номер позиции элемента в списке, тем глубже в дереве расположена соответствующая этому элементу вершина. Значение этого факта состоит в том, что алгоритмы, в которых требуется доступ к компонентам представления в виде терма, почти всегда должны разбирать этот терм сверху вниз, и поэтому чем глубже находятся нужные компоненты, тем больше усилий необходимо приложить для обеспечения доступа к ним. Таким образом, если

нам нужна программа обработки списка, стратегия которой включает в себя много случайных выборок его элементов, то представление списка в виде терма является, по-видимому, неудачным.

В принципе термы можно использовать также для представления многомерных массивов. Например, с помощью терма вида

$10.20.30.NIL; \quad 40.50.60.NIL; \quad 15.20.25.NIL; \quad NIL$

можно было бы представить матрицу

$$\begin{bmatrix} 10 & 20 & 30 \\ 40 & 50 & 60 \\ 15 & 20 & 25 \end{bmatrix}$$

рассматривая каждую ее строку как список элементов и используя инфиксный функтор; для связывания строк в список, заканчивающийся пустой строкой NIL . Такого рода представление будет, по всей видимости, практически полезным только для тех алгоритмов обработки матриц, в которых элементы одной строки обрабатываются последовательно, а сами строки матрицы также просматриваются одна за другой.

IV.2.2. Программы нахождения следующего элемента

Для того чтобы проиллюстрировать далее действие представления структур данных на стиль программирования, мы вернемся к рассмотрению задачи нахождения следующего элемента в списке. А именно задача состоит в том, чтобы определить, какой элемент t следует непосредственно за элементом C в списке $L = (A, B, C, D, E)$. Ее можно сформулировать в виде целевого утверждения

? $\text{след}(C, t, A.B.C.D.E.NIL)$

Наиболее прямым способом выражения понятия «быть следующим элементом» является, вероятно, такой:

$\text{след}(u, v, y) \text{ если } \exists(u, i, y), \exists(v, i + 1, y)$

В этом утверждении говорится о том, что элементы u и v будут соседними в списке y , если номера их позиций различаются на единицу. Если приведенное выше утверждение предполагается использовать в качестве процедуры в программе нахождений следующего элемента, то необходимо будет написать еще несколько процедур, обрабатывающих два вызова \exists . Заметим, что при активации первого из них будет искааться номер позиции i элемента C в списке L , тогда как при активации второго будет искаться элемент v из L , занимающий позицию с (теперь уже

известным) номером $i + 1$; таким образом, два этих вызова э имеют различные характеристики относительно входных и выходных данных. Довольно легко найти простое множество процедур, которое с точки зрения эффективности является достаточно нейтральным по отношению к этим двум видам входа и выхода. В нем используется предикат $\text{занимает}(u, i, j, z)$, который означает, что элемент u занимает позицию с номером $i - j + 1$ в хвостовом фрагменте (z_j, \dots, z_n) списка z . Ниже приводится получающаяся в результате программа.

Программа 20

```
? след( $C, t, A.B.C.D.E.NIL$ )
    след( $u, v, y$ ) если  $\text{э}(u, i, y), \text{э}(v, i + 1, y)$ 
         $\text{э}(u, i, y)$  если  $\text{занимает}(u, i, i, y)$ 
    зан1:  $\text{занимает}(u, i, i, u.x)$ 
    зан2:  $\text{занимает}(u, i, j, w.x)$  если  $\text{занимает}(u, i, j + 1, x)$ 
```

Процедуры зан1 и зан2 порождают типичную итерацию, в которой индекс j управляет просмотром списка L вплоть до той точки, где будет найден требуемый элемент. Несмотря на то что эти процедуры выполняют процесс поиска очень эффективно, исполнение программы в целом оказывается, к сожалению, неэффективным. Причина этого состоит в следующем. Для того чтобы вычислить номер позиции элемента C в L , первый вызов процедуры э должен с помощью процедуры зан2 разложить список L на последовательные хвостовые фрагменты $A.B.C.D.E.NIL$, $B.C.D.E.NIL$ и $C.D.E.NIL$; само по себе это разложение приемлемо, поскольку оно равнозначно линейному поиску элемента C в L . Однако второй вызов процедуры э , который ищет четвертый элемент списка L , должен еще раз образовать те же самые фрагменты. Таким образом, мы расплачиваемся за определение понятия «быть следующим элементом» в виде двух независимо обрабатываемых вызовов э , каждый из которых несет бремя выполнения косвенного доступа к представлению списка L посредством структурированного терма.

Один из способов получения более приемлемого поведения в период исполнения, не принося при этом в жертву используемую в программе 20 процедуру след, состоит в том, чтобы обходиться без всех оставшихся процедур, а просто полагаться на специальные процедуры э , встроенные в интерпретатор. В результате этого терм $A.B.C.D.E.NIL$ был бы представлен с помощью внутреннего массива, и решение вызова $\text{э}(v, 4, A.B.C.D.E.NIL)$ в программе было бы найдено тогда за один шаг посредством осуществления прямого доступа к четвертой ячейке упомянутого массива. Встроенные процедуры прямого доступа для обработки списков оказываются очень полезными средствами,

поскольку обработка списков постоянно требуется для вычислительного решения задач на ЭВМ. В логике, однако, допускается так много других видов термов, что практически невозможно встроить в интерпретатор процедуры прямого доступа для каждого из них, и, стало быть, общая проблема организации доступа к термам все же остается.

Еще один способ повышения эффективности заключается в изменении определения отношения «быть следующим элементом». Можно использовать, например, определение, уже встречавшееся в предыдущих главах:

$$\begin{aligned} \text{след}(u, v, u.v.x) \\ \text{след}(u, v, w.y) \text{ если } \text{след}(u, v, y) \end{aligned}$$

Эти процедуры эффективным образом объединяют отдельные вызовы `э` из программы 20, вследствие чего разложение входного списка выполняется не два раза, а один. Они являются чрезвычайно эффективными, однако интуитивно не так очевидны, как процедура `след` из программы 20.

IV.2.3. Программы для задачи о палиндроме

Палиндром — это список, который читается вперед точно так же, как и назад, т. е. список, совпадающий со своим обратным. Задача, состоящая в том, чтобы определить, является ли данный список палиндромом, предоставляет целый ряд возможных вариантов алгоритма и стиля программирования. Допустим, что у нас имеется список (A, B, C, B, A) . Один из способов формализации этой задачи — без всяких премудростей интерпретировать только что приведенную ее спецификацию:

$$\begin{aligned} ? \text{ палин}(A.B.C.B.A.NIL) \\ \text{палин } (x) \text{ если } \text{обратить}(x, x) \\ \text{и какие-либо процедуры для предиката } \text{обратить} \end{aligned}$$

Эффективность этого подхода в значительной степени определяется выбором процедур обращения списка. В разделе III.2 предыдущей главы мы рассматривали две возможности. Первая из них заключалась в использовании процедур

$$\begin{aligned} \text{обратить}(NIL, NIL) \\ \text{обратить}(u.x', y) \text{ если склеить}(y', u.NIL, y) \\ \text{обратить}(x', y') \end{aligned}$$

Вызов указанных процедур при исполнении нашей программы имеет целью показать, что список $u.x' = A.B.C.B.A.NIL$ является обратным по отношению к списку $y = A.B.C.B.A.NIL$. Это

будет сделано посредством (i) вычеркивания первого элемента $u = A$ из $u.x'$, в результате чего останется список $x' = B.C.B.A.NIL$, (ii) подтверждения того, что элемент $u = A$ является последним в списке y , и затем вычеркивания его из y , в результате чего останется список $y' = A.B.C.B.NIL$, и, наконец, (iii) посредством установления того, что список x' — обратный по отношению к списку y' .

Решение второй из этих трех задач осуществляется при помощи вызова **склеить** и оказывается поэтому главным источником неэффективности, поскольку не существует таких процедур **склеить**, которые решали бы данный вызов за один шаг — наоборот, в любом случае они должны будут шаг за шагом раскладывать список y и одновременно с этим шаг за шагом составлять список y' . Еще хуже то, что при каждом вызове итеративной процедуры **обратить** будет выполняться разложение (при помощи процедуры **склеить**) тех фрагментов, которые совершенно точно так же раскладывались в результате предыдущих вызовов; таким образом, исполнение нашей программы содержит слишком много излишних действий. В результате всего этого время, требуемое для выполнения алгоритма, находится по крайней мере в квадратичной зависимости от длины входного списка. Основную причину такого плохого поведения можно обнаружить в невозможности получения прямого доступа к последнему элементу терма, построенного из точек и константы *NIL*.

Другой возможный подход состоит в использовании следующих процедур обращения:

```
обратить( $x, y$ ) если обратить*(NIL,  $x, y$ )
обратить*( $y, NIL, y$ )
обратить*( $x_1, u.x_2, y$ ) если обратить*( $u.x_1, x_2, y$ )
```

Как мы уже видели в гл. III, эти процедуры ведут себя намного эффективнее, хотя их логика более трудна для понимания. Время работы алгоритма, получающегося в результате использования указанных процедур, будет зависеть лишь линейно от длины входного списка.

Еще более радикального улучшения можно добиться за счет изменения исходной спецификации задачи. А именно, мы будем теперь говорить, что список является палиндромом, если его первая половина оказывается обращением второй. При таком подходе уже нет необходимости заниматься обращением всего списка, что имело место в предыдущих методах. Приводимая ниже программа основана на этой идеи, которая позволяет к тому же обходиться без явных процедур обращения.

Программа 21

```
? палин(A.B.C.B.A.NIL)
палин(x) если палин*(x,NIL)
палин*(z,z)
палин*(u.z,z)
палин*(u.z',z) если палин*(z',u.z)
```

Здесь предикат **палин*** (z_1, z_2) означает, что палиндром получается в результате присоединения списка z_1 к обращению списка z_2 . Получаемое в результате исполнение оказывается очень хорошим (упражнение: исполните эту программу вручную). Данный алгоритм примерно в два раза быстрее приведенного выше алгоритма, использующего процедуры *обратить**. Та неэффективность, которая еще остается, возникает главным образом из-за затрат, связанных с унификацией различных структурированных термов. Эти затраты находятся вне пределов контроля со стороны программиста; они будут определяться заложенным в интерпретаторе конкретным способом обработки термов. Разумеется, усилия, требуемые для выполнения унификации в ходе исполнения программы, могут быть в значительной степени уменьшены, если интерпретатор оказывается в состоянии проводить предварительный интеллектуальный анализ входной программы и, таким образом, выбирать способы эффективной реализации намерений программиста. Эта возможность, которая относится к общим вопросам оптимизации в период компиляции, представляется в особенности привлекательной тем, что она позволяет писать программы, основанные на термах относительно высокого уровня, зная при этом, что конкретная реализация программы будет организована так, чтобы избежать значительных расходов, связанных с доступом к данным и унификацией. Программа 21, например, могла бы быть реализована с помощью лишь двух линейных массивов, соответствующих двум аргументам предиката **палин***. Третья процедура **палин*** была бы тогда эффективно скомпилирована в множество машинных операций, которые быстро переносили бы элемент из первого массива во второй.

Если имеющийся в распоряжении интерпретатор не способен выполнять какую-либо существенную оптимизацию, то для усовершенствования программы 21 необходимо избавиться от представления посредством структурированных термов. Вместо этого можно прибегнуть к представлению входного списка в виде фактов и составить процедуры, моделирующие алголоподобное поведение, т. е. включающие управление на низком уровне циклами и индексами. Иллюстрация подобного подхода дается в разд. IV.3.

IV.2.4. Контроль соответствия типов

Когда большие структуры данных собираются при помощи какого-либо метода, подверженного ошибкам (например, при помощи ручного ввода), может оказаться выгодным осуществлять некоторый контроль этих данных перед выполнением основных действий, связанных с их обработкой. Хотя контроль мог бы включать в себя много различных видов тестов данных, наиболее фундаментальным из них является контроль соответствия типов, посредством которого данные проверяются на структурное соответствие имеющемуся в виду типу данных.

Допустим в качестве примера, что мы хотим осуществить ввод дерева z , подобного тому, которое изображено на рис. IV.1c, и затем каким-то образом его обработать. Если бы контроль соответствия типов помещался между стадией ввода и стадией обработки, то целевое утверждение можно было бы сформулировать следующим образом:

? ввести(z), дерево(z), обработать(z, z'), вывести(z')

Здесь подчеркнутый вызов дерево (z) предназначен для выполнения контроля соответствия входных данных для z типу «дерево». Если результат этого контроля будет отрицательным, то исполнение программы будет тем самым избавлено от возможного порождения более дорогостоящего и менее понятного неудачного результата при решении вызова обработать (z, z'). Сам контроль можно было бы осуществлять с помощью процедур

дерево(z) если $z = TIP$
дерево(z) если $z = t(z_1, z_2)$, дерево(z_1), дерево(z_2)

Контроль соответствия типов может быть связан с довольно большими расходами в период исполнения и поэтому имеет смысл обнаруживать ошибки как можно раньше. В приведенном выше примере вызов ввести (z) можно было бы реализовать как приглашение пользователю вводить данные с некоторого устройства ввода. Если вводимая структура данных оказывается большой и чувствительной к нескольким ошибкам, то было бы очень неэффективно сначала вводить ее всю и лишь затем производить контроль соответствия типов; гораздо лучше продолжать ввод данных только до тех пор, пока не встретится первая ошибка. Исходя из этих соображений, мы получим значительное преимущество, если стратегия управления, заложенная в интерпретаторе, будет включать в себя возможность работать в *сопрограммном режиме*. В IC-Прологе, например, поддерживаются сопрограммная стратегия «производитель-потребитель», которой можно задать такие инструкции, чтобы заставить вызов ввести (z) поставлять очередную порцию данных только в том

случае, когда процесс обработки вызова дерево (z) готов потребить их для продолжения контроля соответствия типов. В действительности поведение могло бы быть следующим. Вызовом дерево (z) делается начальный запрос к вызову ввести (z), требующий введения некоторых данных, в результате чего последний вызов активируется и выдает приглашение пользователю начинать ввод; пользователь печатает, скажем, $z := t(x, y)$, и в этот момент управление снова возвращается к вызову контроля соответствия типов и производит некоторое количество необходимых для решения этого вызова шагов, например:

```
? дерево(z)
? дерево(t(x, y))
? t(x, y) = t(z1, z2), дерево(z1), дерево(z2)
? дерево(x), дерево(y)
```

Далее процесс контроля соответствия типов продолжаться не может до тех пор, пока не будут введены новые конкретные данные, поэтому управление вновь возвращается к процессу ввода и запрашивает пользователя, который вводит теперь, скажем, $x := t(TIP, TIP)$ и, стало быть, заполняет еще одну компоненту структуры данных. Она в свою очередь подвергается контролю соответствия типов, и, наконец, пользователь заполняет последнюю компоненту, скажем $y := TIP$, которая затем также проверяется. Таким образом, вся структура данных $z := t(t(TIP, TIP), TIP)$ введена по этапам, разделяемым контролем соответствия типов, так что ошибка на каком-либо этапе будет обнаружена до того, как произойдет переход к следующему этапу. Заметим, наконец, что логические программы способны обрабатывать *частично определенные данные* в виде термов, которые содержат переменные, не связанные никакими другими термами. Выше мы видели, что частично определенная структура $t(x, y)$ могла передаваться процедурам контроля соответствия типов и частично обрабатываться до того, как стали известными конкретные значения переменных x или y . Этот вид поведения не имеет прямых аналогов во многих традиционных языках программирования.

IV. 3. Представления посредством фактов

В представлениях посредством фактов используется то обстоятельство, что сами логические процедуры могут служить в качестве описаний компонент структур данных. Их можно применять для описания всех типов данных, например списков, матриц и деревьев, которые иначе можно было бы представить в виде термов; общее свойство этих типов данных заключается

в том, что они имеют регулярную структуру. Программы, обрабатывающие представления таких видов данных посредством фактов, часто должны обращаться за помощью к процедурам доступа довольно низкого уровня, которые регулируют использование индексов, указателей и т. п. Кроме того, представления посредством фактов оказываются также в высшей степени полезными для описания совершенно нерегулярных наборов данных, таких как базы данных, чье описание с помощью термов было бы чрезмерно громоздким.

IV.3.1. Общие принципы

Если мы описываем некоторое отношение, формулируя правило порождения его элементов, то говорят, что это отношение определяется *интенсионально*. Так, например, процедура

$$\exists(u, i, L) \text{ если } 1 \leq i, i \leq 3, u = i * 10$$

служит в качестве интенсионального определения отношения \exists ; для того чтобы найти элементы \exists , требуется применять данное определение. В противоположность этому множество фактов

$$\begin{aligned} &\exists(10, 1, L) \\ &\exists(20, 2, L) \\ &\exists(30, 3, L) \end{aligned}$$

составляет *экстенсиональное* определение отношения \exists , поскольку в нем явно указываются все его элементы и, стало быть, нет необходимости применять какое-либо правило для их нахождения. Экстенсиональное определение и является как раз наиболее простым видом представления посредством фактов.

Заметим, что в каждом из фактов в приведенном выше примере имеется ссылка на имя L , которое служит там в качестве имени представляемой структуры данных. Эти ссылки нужны лишь как средство, позволяющее различать структуры данных, когда мы имеем дело с несколькими из них одновременно.

Часто встречается ситуация, когда для описания какой-то конкретной структуры данных требуется использовать несколько отношений, и поэтому на ее имя должны делаться ссылки во всех фактах для каждого из этих отношений. Так, в случае представления списка мы, как правило, сопровождали бы факты \exists еще одним фактом

$$\text{длина}(L, 3)$$

в котором определяется длина списка L . Заметим, что программисты, пишущие на традиционных языках, делают почти то же самое; они хранят список в массиве L и, кроме того, хранят его

длину в качестве значения некоторой переменной n , чтобы всегда иметь ее под рукой для выполнения таких операций, как

for $i = 1$ to n **do обработать** $L(i)$

При описании реальных баз данных может потребоваться довольно много различных отношений. В качестве простого примера рассмотрим базу данных, используемую для управления предварительным заказом мест в гостинице. Каждый номер в гостинице будет характеризоваться некоторым идентификатором, выбираемым, быть может, среди чисел 1, 2, 3, ... и т. д.; вместимостью, скажем, одноместный (O) или двухместный (D); номером этажа, на котором он расположен; наличием душа, а также множеством периодов времени, на которые он уже забронирован. Состояние гостиницы в каждый момент времени описывается в некоторой степени нерегулярной структурой данных. Ее представление в виде фактов могло бы выглядеть следующим образом:

этаж(1, ПЕРВЫЙ)	вместимость(1, D)	душ(1, НЕТ)
этаж(2, ПЕРВЫЙ)	вместимость(2, O)	душ(2, ДА)
этаж(3, ВТОРОЙ)	вместимость(3, D)	душ(3, ДА)
этаж(4, ВТОРОЙ)	вместимость(4, D)	душ(4, ДА)
этаж(5, ТРЕТИЙ)	вместимость(5, O)	душ(5, НЕТ)
забронирован(1, НОЯБРЬ19, НОЯБРЬ22)		
забронирован(2, ДЕКАБРЬ14, ДЕКАБРЬ14)		
забронирован(2, ФЕВРАЛЬ04, МАРТ03)		
забронирован(4, ДЕКАБРЬ14ДЕКАБРЬ15)		

К этой базе данных можно было бы обращаться с запросами вида

**? вместимость(r, D), душ(r, DA),
наличие ($r, ДЕКАБРЬ13, ДЕКАБРЬ16$)**

т. е. какой двухместный номер r имеет душ и свободен в период с 13 по 16 декабря? База данных поддерживалась бы различными процедурами, способными искать информацию или вносить изменения в соответствии с ответами на запросы или исправлениями. Если бы мы имели дело с базой данных для агентства, принимающего предварительные заказы мест в гостиницах, то наши факты должны были бы, кроме того, содержать имена различных гостиниц, которые обслуживаются этим агентством.

IV.3.2. Контроль соответствия типов

Для того чтобы оценить структурную правильность представлений посредством фактов, иногда может потребоваться подвергать их контролю соответствия типов. Так, например, случайное

искажение фактов, представляющих наш список L , могло бы видоизменить их до следующего состояния:

$$\begin{array}{ll} \text{э}(10, -1, L) & \text{длина}(L, 2) \\ \text{э}(20, 2, L) & \text{длина}(L, 4) \\ \text{э}(30, 2, L) & \end{array}$$

которое больше не соответствует предполагаемому списковому типу данных. Поэтому в программе, предназначеннной для обработки L , можно было бы сначала выполнить контроль соответствия типов с помощью вызова списка (L), используя при этом процедуры, которые проверяли бы, что длина n списка L неотрицательна и единственна и что на каждой его позиции с целым номером i , заключенным в пределах от 1 до n , находится в точности один элемент. Эти процедуры могли бы выглядеть следующим образом:

$$\begin{aligned} \text{список}(x) \text{ если } & \text{длина}(x, n), n \geq 0, \sim \text{еще-длина}(x, n), \\ & \text{элементы}(x, 1, n) \\ \text{элементы}(x, i, n) \text{ если } & i > n \\ \text{элементы}(x, i, n) \text{ если } & i \leq n, \text{э}(u, i, x), \\ & \sim \text{еще-элемент}(u, i, x), \text{элементы}(x, i + 1, n) \\ \text{еще-длина}(x, n) \text{ если } & \text{длина}(x, n'), n \neq n' \\ \text{еще-элемент}(u, i, x) \text{ если } & \text{э}(u', i, x), u \neq u' \end{aligned}$$

В серьезных программах для организации баз данных также может потребоваться выполнять проверки правильности данных, особенно во время их обновления. Правила, регулирующие правильность данных, называются *ограничениями целостности*; обычный контроль соответствия типов является как раз простым примером более общего процесса, который проверяет, удовлетворяет ли какой-либо набор данных своим ограничениям целостности. В приведенном ранее примере с заказом мест в гостинице одним из возможных ограничений могло бы быть следующее утверждение, которое предохраняет от накладок при заказе номеров при условии, конечно, что данные всегда ему удовлетворяют.

$d_2 < d_3$ если забронирован(r, d_1, d_2), забронирован(r, d_3, d_4), $d_1 < d_3$

Такого рода утверждения можно помещать в системы логических баз данных и специальным образом их вызывать с целью обнаружения противоречивости или неполноты данных, или для того чтобы предотвратить возникновение подобных ситуаций.

IV.3.3. Индексирование

В типичном логическом интерпретаторе процедуры входной программы будут храниться в памяти машины, для чего используется какая-либо схема *индексирования*. Эта схема применя-

ется в период компиляции для эффективной классификации процедур в соответствии как с их именами, так и со структурным и лексическим содержанием их формальных параметров; она устанавливает затем указатели местонахождения в памяти каждого класса процедур, выделенного в ходе упомянутой классификации. Информация об указателях классов собирается в легкодоступной форме в одной или нескольких таблицах индексов, которые также хранятся в памяти и к которым можно будет обращаться в период исполнения для получения быстрого доступа к хранимым процедурам. Более точно, при активации какого-либо вызова интерпретатор сравнивает его имя и фактические параметры с входами в таблицу и таким образом находит указатель на ту область памяти, где содержится потенциально отвечающие вызову процедуры; этот метод, очевидно, более эффективен, чем осуществление слепого поиска в беспорядочном наборе хранящихся в памяти процедур. Конкретный вид схемы, используемой для сопоставления заголовкам процедур ссылок на ячейки памяти, будет зависеть от интерпретатора, который может комбинировать классификацию по умолчанию со специальной информацией относительно индексирования, предоставляемой ему программистом.

Особое значение индексирование имеет для эффективной реализации представлений данных посредством фактов. Применяя обсуждавшиеся выше идеи к нашему представлению списка L в виде фактов, мы можем построить основную таблицу индексов для отношения \exists , содержащую указатель входа в область памяти, отведенную всем тем процедурам \exists , которые имеют L в качестве третьего параметра. Среди них можно выделить те процедуры, второй параметр которых является целым положительным числом, и разместить их в соответствующей зоне внутри указанной области. Доступ к ним можно осуществлять при помощи вспомогательной таблицы индексов, предназначеннной для классификации по второму параметру.

Когда в ходе исполнения программы активируется вызов вида $\exists(u, 2, L)$, интерпретатор сравнивает входы таблицы индексов с символами \exists , 2 и L (которые служат, стало быть, в качестве ключей для поиска) для того, чтобы найти ссылку, определяющую местонахождение в памяти потенциально отвечающих на этот вызов процедур. В нашем примере таким способом будет обнаружен единственный факт $\exists(20, 2, L)$. Несколько более сложную схему можно было бы применить, если бы интерпретатор мог действовать во время компиляции на основании информации, касающейся типа данных множества фактов \exists , ибо в этом случае вспомогательная таблица индексов могла бы просто связывать каждый факт \exists , имеющий вторым параметром i , с i -м сегментом области памяти, в котором в свою оче-

редь не хранилось бы ничего, кроме первого параметра этого факта, т. е., другими словами, только значение i -го элемента списка L . Такая схема, очевидно, соперничала бы с традиционной скомпилированной программой, которая выбирает значение $L(2)$, используя индекс 2 в качестве величины сдвига от базового адреса списка L в памяти машины.

Наличие схем индексирования является существенным предварительным условием для принятия точки зрения, согласно которой представления посредством фактов делают компоненты структур данных легкодоступными. Подобные схемы побуждают нас, к примеру, сформулировать задачу о следующем элементе из разд. IV.2.2 в таком стиле:

$$\begin{aligned} ? \text{ след}(C, t, L) \\ \text{след}(u, v, y) \text{ если } \exists(u, i, y), \exists(v, i + 1, y) \\ \exists(A, 1, L) \exists(B, 2, L) \exists(C, 3, L) \\ \exists(D, 4, L) \exists(E, 5, L) \end{aligned}$$

Хороший интерпретатор способен будет реализовать факты \exists как представление списка L в виде линейного массива (так, как это описано выше) и получать прямой доступ к его элементам, используя индексированный поиск.

IV.3.4. Массивы и индексы

Списки, матрицы и другие структурированные подобно массивам множества данных в особенности пригодны для представлений посредством фактов. В частности, индексированные хранение и выборка фактов обеспечивают эффективную реализацию алгоритмов, управляемых с помощью контроля индексов. Мы вновь обратимся здесь к задаче о палиндроме из разд. IV.2.3 для того, чтобы посмотреть, как логический программист может выразить управляемый при помощи индексов итеративный процесс просмотра элементов в линейном массиве. Входные данные включают в себя список $L = (A, B, C, B, A)$, имеющий длину $n = 5$, и представлены они обычным образом в виде фактов \exists и длина.

Мы будем пользоваться следующим алгоритмом. На каждом шаге итерации по двум индексам i и j , имеющим начальные значения 1 и n соответственно, выбираются два элемента $L(i)$ и $L(j)$, которые у палиндрома должны совпадать. Итеративный процесс, идя с обоих концов списка L , постепенно продвигается внутрь путем одновременного увеличения значения индекса i и уменьшения значения индекса j . Этот процесс заканчивается, когда i и j «пересекутся», т. е. когда будет справедливо неравенство $i > j$. Приведенный алгоритм не слишком отличается от алгоритма, получаемого из основанной на представлении

в виде термов программы 21, однако он реализуется более эффективно, поскольку элементы списка L можно выбирать и сравнивать более прямо. В традиционной нотации наш алгоритм можно представить следующей программой:

```

 $n := \text{длина списка } x; i := 1; j := n;$ 
while  $i \leq j$  do
    begin if  $x(i) = x(j)$  then begin  $i := i + 1;$ 
                                 $j := j - 1;$ 
                                end
        else НЕУДАЧА;
    end; УСПЕХ

```

Для того чтобы выразить все это на языке логики, мы выберем имя, скажем **палин****, обозначающее итерацию **while ... do**. Точнее, предикат **палин**** (x, i, j) означает, что список $(x(i), \dots, x(j))$ является палиндромом, а x, i и j суть аргументы предиката **палин****, поскольку эти переменные определяют каждый шаг итерации. Начальные значения i и n присваиваются переменным i и j в результате первого вызова процедуры **палин**** после того, как будет найдена длина n списка L . Ниже приводится полный текст программы.

Программа 22

```

? палин( $L$ )
    палин( $x$ ) если длина( $x, n$ ), палин**( $x, 1, n$ )
    палин**( $x, i, j$ ) если  $i > j$ 
    палин**( $x, i, j$ ) если  $i \leq j$ ,  $\exists(u, i, x), \exists(u, j, x),$ 
                                палин**( $x, i + 1, j - 1$ )
     $\exists(A, 1, L) \exists(B, 2, L) \exists(C, 3, L)$ 
     $\exists(B, 4, L) \exists(A, 5, L) \text{длина}(L, 5)$ 

```

Каждый обход вокруг цикла начинается с вызова первой процедуры **палин****, предназначеннной для проверки условия завершения; если этот вызов оказывается неудачным, то тогда используется вторая процедура **палин**** для проверки следующей выбранной пары элементов. С логической точки зрения эта вторая процедура должна также включать проверку $i \leq j$, которая несколько снижает эффективность исполнения программы. Ее, однако, можно устраниТЬ путем применения конструкции **если — то — иначе** (упоминавшейся в гл. III в связи с программой 18), заменивая обе процедуры **палин**** единственным условным утверждением

```

палин**( $x, i, j$ ) если  $i \leq j$  то  $\exists(u, i, x),$ 
                                 $\exists(u, j, x), \text{палин**}(x, i + 1, j - 1)$ 
                                иначе выход

```

Стиль, в котором написана программа 22, можно, очевидно, распространить и на рассмотрение итераций в многомерных массивах.

IV.4. Обработка данных в виде фактов

Обычно мы требуем, чтобы представления структур данных были пригодными как для ввода, так и для вывода. Ни сами эти понятия, ни реализация указанного требования не вызывают никаких трудностей в связи с представлениями данных посредством структурированных термов, свойства которых по отношению к вводу и выводу уже обсуждались в гл. II. Однако манипулирование данными, представленными в виде фактов, оказывается не столь простым и требует специальных методов программирования.

IV.4.1. Имена как выходные данные

В целевом утверждении программы 22 спрашивается о свойствах объекта с заданным именем L . В силу недетерминированности логических процедур по отношению к вводу и выводу их можно использовать также для нахождения и вывода имен тех объектов, которые удовлетворяют заданным свойствам.

Допустим в качестве примера, что у нас имеется база данных, содержащая много списков, представленных в виде фактов, и пусть требуется найти все списки x , y и z , такие что z является векторной суммой x и y . Согласно определению векторной суммы $z = x \oplus y$, каждый i -й элемент списка z является суммой i -х элементов списков x и y ; при этом предполагается, что списки x , y и z имеют одинаковую длину. Приводимая ниже программа выполняет это задание непосредственно.

Программа 23

```
? сумма( $x, y, z$ )
  сумма( $x, y, z$ ) если длина( $x, n$ ), длина( $y, n$ ),
    длина( $z, n$ ), сумма*( $x, y, z, 1, n$ )
  сумма*( $x, y, z, i, n$ ) если  $i > n$ 
  сумма*( $x, y, z, i, n$ ) если  $i \leq n$ , э( $u, i, x$ ),
    э( $v, i, y$ ), э( $w, i, z$ ),
    сложить( $u, v, w$ ), сумма*( $x, y, z, i + 1, n$ )
    а также факты длина и э, определяющие все
    списки в нашей базе данных
```

При вызове процедуры сумма сначала будут выбраны некоторые списки x , y и z одинаковой длины, а затем будут вызваны процедуры сумма*, которые проверят, удовлетворяют ли эти списки

требуемому отношению $z = x \oplus y$. В результате успешных вычислений целевым переменным x , y и z будут присвоены различные имена списков. Можно сказать, что программа выдает на выходе списки в ограничении смысле, а именно она сообщает только их имена. В том же самом смысле конкретные имена, которые упоминаются в целевом утверждении, можно было бы рассматривать как списки, задаваемые в качестве входных данных программы.

IV.4.2. Факты как выходные данные: метод, используемый в Прологе

Совершенно иное понятие выходных данных возникает, когда мы рассматриваем возможность такого исполнения программы, в ходе которого будут порождаться новые факты, представляющие данные. Те факты, что заданы в программе изначально, могут в этом случае считаться входными данными, тогда как факты, возникающие в результате ее исполнения, составляют выходные данные. Существует несколько способов порождения фактов в качестве выходных данных, и все они требуют специальной адаптации стандартного интерпретатора.

Метод, применяемый в классическом (т. е. марсельском) Прологе, основывается на том, что программист может писать вызовы вида **факт** (t), где t — структурированный терм. Каждый такой терм имеет тот же самый синтаксис, что и факты, и поэтому указанный вызов можно читать как директиву «имеет место факт t », т. е. «добавить к тексту программы факт t ». При активации этот вызов решается непосредственно с помощью встроенной процедуры, которая добавляет факт t к остальным утверждениям программы. Допустим к примеру, что t — это терм $\text{э}(20, 2, C)$. Тогда директива **факт** (t) интерпретирует функцию э терма t как имя процедуры э , а его аргументы 20 , 2 и C — как формальные параметры процедуры э . Таким образом, результатом решения вызова **факт** ($\text{э}(20, 2, C)$) является добавление к программе нового факта $\text{э}(20, 2, C)$.

Использование директивы **факт** можно проиллюстрировать на следующей задаче: по заданным представлениям списков A и B в виде фактов образовать представление посредством фактов списка $C = A \oplus B$. Эту задачу можно решить с помощью проводимой ниже программы 24, которая получается из программы 23 просто за счет того, что два вызова из ее процедур помещаются в качестве аргументов директив **факт** ().

Программа 24

```
? сумма(А, В, С)
    сумма(Х, Y, Z) если длина(Х, N), длина(Y, N),
        факт(длина(Z, N)), сумма*(Х, Y, Z, I, N)
```

сумма*(x, y, z, i, n) если $i > n$
 сумма*(x, y, z, i, n) если $i \leq n, \exists(u, i, x), \exists(v, i, y),$
 сложить(u, v, w), факт($\exists(w, i, z)$),
 сумма*($x, y, z, i + 1, n$)

а также факты *длина* и \exists , определяющие списки *A* и *B*

Если входными списками являются, скажем, $A = (3, 12, 23)$ и $B = (7, 8, 7)$, то в результате исполнения программы 24, будут образованы факты

$\exists(10, 1, C)$	$\exists(30, 3, C)$
$\exists(20, 2, C)$	длина($C, 3$)

представляющие список $C = A \oplus B$.

Поскольку в активируемых вызовах факт термы могут строиться произвольным образом за счет распределения данных на предыдущих шагах исполнения программы, механизм, основанный на директивах факт, позволяет программам расширять самих себя до любой желаемой программистом степени. Более того, как только будет образован новый факт, к нему могут обращаться последующие вызовы, и таким образом он будет оказывать влияние на дальнейший ход исполнения программы. Позднее мы рассмотрим пример подобного поведения.

Если интерпретатору не дано никаких других инструкций, то ему, по всей видимости, следовало бы (ради соблюдения непротиворечивости философии реализации) отменять (вычеркивать) факты, когда на его пути в процессе возврата встречаются шаги, которые привели к их появлению точно так же, как он отменяет в ходе возврата присваивания переменным. По крайней мере их следует, вообще говоря, отменять, когда интерпретатор осуществляет возврат по неудачному вычислению. В некоторых реализациях программисту предоставлен целый ряд директив, дающих полный контроль над порождением и удалением фактов в период исполнения.

Директива факт является довольно мощной, и ее легко реализовать. Однако недостаток ее заключается в том, что, хотя операционное действие вызова факт может быть достаточно очевидным при рассмотрении его независимо от контекста, логический смысл исполнения программы в целом может оказаться неясным. Это средство является на самом деле слишком сильным, если его применять без всяких ограничений. Вообще говоря, оно затрудняет понимание взаимосвязи между входной программой и вычисленными с ее помощью «решениями». Критики утверждают, что ничем не ограниченное использование директивы факт приводит к отклонению от чистого логического программирования в направлении формализма, имеющего со-

всем другую семантику. Тем не менее в ряде случаев можно искусно обойти эту критику, объясняя исполнение нестандартной программы Р (содержащей вызовы факт), исходя из нормального исполнения некоторой тесно связанной с ней стандартной программы Q. Обоснование программы Р базируется в этом случае на том утверждении, что каждое «решение», вычисляемое с помощью Р, с необходимостью является логически правильным решением Q, и что исполнение программы Р, следовательно, есть лишь удобный способ реализации исполнения программы Q.

Чтобы проиллюстрировать это, предположим, что Р есть наша программа 24 из предыдущего примера. Если мы удалим из Р все директивы факт (), но оставим нетронутыми их аргументы в программе и добавим к Р все те факты, которые порождаются исходными вызовами факт, то в результате мы получим следующую стандартную программу Q:

Программа 25

```
? сумма(A,B,C)
  сумма(x,y,z) если длина(x,n), длина(y,n),
    длина(z,n), сумма*(x,y,z,I,n)
  сумма*(x,y,z,I,n) если I > n
  сумма*(x,y,z,I,n) если I ≤ n, э(u,I,x), э(v,I,y),
    сложить(u,v,w), э(w,I,z),
    сумма*(x,y,z,I+1,n)
  а также факты длина и э, представляющие списки A, B и C
```

По существу это та же программа 23, но с более конкретным целевым утверждением. Оказывается в этом случае, что если (нестандартное) исполнение программы Р «решает» целевое утверждение сумма (A, B, C), то его решает и (стандартное) исполнение программы Q. Стало быть, Р служит просто в качестве удобной версии Q, которая используется тогда, когда факты, представляющие список C, являются выходными данными, а не входными.

Таким способом может быть очень трудно обосновать программу Р, в которой в последующем происходят обращения к фактам, появившимся в результате использования вызовов факт. Задача усложняется, когда в программе Р помимо вызовов факт содержатся еще вызовы «отменить», которые специфицируют удаление фактов в ходе вычислений и которые также имеются в Прологе. В этом случае может оказаться так, что не существует никакой стандартной программы Q, прямо и понятно связанной с Р, чье исполнение могло бы объяснить исполнение программы Р. Трудности еще более возрастают при попытках обосновать программы, в которых наряду с указанными дирек-

тивами содержатся также отрицательные вызовы, решаемые посредством невыводимости, поскольку динамическая самомодификация программы приводит к изменению множества невыводимых из нее предложений. По сути дела общая проблема, касающаяся семантики классического Пролога, возникает не столько из-за действия его нестандартных возможностей, рассматриваемых изолированно друг от друга, сколько вследствие их потенциального взаимодействия.

Несмотря на то что директиву **факт** можно использовать как генератор фактов методом «грубой силы», ее можно применять также и с некоторыми ограничениями, которые сохраняют логическую семантику. Так, например, порождение логически правильно вытекающего из утверждений программы факта на основании успешного выхода из процедуры достигается с помощью конструкций вида

$$\text{p}(x) \text{ если... факт}(\text{p}(x))$$

Или, что эквивалентно, логически правильно вытекающий факт, получаемый в результате решения конкретного вызова, скажем **p(x)**, может быть порожден конструкциями вида

$$\text{q если..., p}(x), \text{ факт}(\text{p}(x)), \dots$$

Стандартный Пролог позволяет оформлять такого рода механизмы в виде процедур метауровня. Например, процедура

$$\text{лемма}(z) \text{ если } z, \text{ факт}(z)$$

строит логически правильно вытекающий факт из каждого разрешимого вызова, вводимого посредством метапеременной **z**. С ее помощью предыдущую процедуру можно записать короче:

$$\text{q если..., лемма}(\text{p}(x)), \dots$$

Это приводит нас к следующему разделу, в котором обсуждается порождение лемм, позволяющее избежать семантических опасностей неограниченного применения директивы **факт**.

IV.4.3. Факты как выходные данные: порождение лемм

В этом новом методе порождения фактов используется то обстоятельство, что после решения какого-либо вызова его можно добавить в качестве факта к исполняемой программе, не изменяя при этом ее логического содержания. Допустим, например, что во входной программе присутствует вызов **э(w, i, z)**, который со временем обычным образом активируется; к этому моменту переменным **i** и **z** могли уже быть присвоены значения **2** и **C** соответственно. Если активированный вызов **э(w, 2, C)** решается, в результате чего переменной **w** присваивается, ска-

жем, значение 20, то тем самым устанавливается, что факт $\exists(20, 2, C)$ логически следует из процедур программы. Стало быть, факт $\exists(20, 2, C)$ является просто утверждением о проблемной области, которое вытекает из уже имеющихся в программе утверждений. Добавление его к программе, следовательно, ни в коей мере не изменит того, что программа говорит о рассматриваемой задаче, и поэтому не может повлиять на множество вычисляемых с ее помощью решений.

Добавление доказанных в период исполнения фактов может повысить эффективность исполнения программы, поскольку некоторые из последующих вызовов могут в этом случае непосредственно решаться путем обращения к новым фактам, а не к тем процедурам (что было бы необходимо при стандартном исполнении), которые использовались для их установления. Когда такие факты вызываются, они выполняют функции лемм, т. е. промежуточных теорем, которые были получены в ходе процесса доказательства. Поэтому мы и называем обсуждаемый метод *порождением лемм*.

Рассмотрим в качестве примера задачу, состоящую в том, чтобы распечатать сначала список C , а затем его обращение, причем список C должен быть вычислен как векторная сумма списков $A = (3, 12, 23)$ и $B = (7, 8, 7)$. Мы начнем с того, что посмотрим на приводимую ниже стандартную программу, предназначенную для решения этой задачи.

Программа 26

```
? длина(C, n), печать*(C, I, n), печать***(C, I, n)
  печать*(z, i, j) если i > j
  печать*(z, i, j) если i ≤ j, э(w, i, z),
    печать(w), печать*(z, i + 1, j)
  печать***(z, i, j) если i > j
  печать***(z, i, j) если i ≤ j, э(w, j, z),
    печать(w), печать***(z, i, j - 1)
  длина(C, n) если длина(A, n), длина(B, n)
  э(w, i, C) если э(u, i, A), э(v, i, B), сложить(u, v, w)
  а также факты длина и э, представляющие списки A и B
```

Здесь предполагается, что вызов печать (w), обращаясь к встроенной процедуре, передает свой параметр на печатающее устройство. Процедуры печать* и печать*** просто ведут итерации соответственно вперед и назад по списку для того, чтобы распечатать его элементы в естественном и обратном порядке. Последние две процедуры, приведенные в программе, предназначены для определения длины списка C и его элементов посредством косвенного доступа. Таким образом, итерация, порождаемая в ходе решения последующего вызова печать*($C, I, 3$),

вычисляет каждый i -й элемент C путем сложения i -х элементов списков A и B . Вторая итерация, порождаемая в ходе решения вызова `печать** $(C, 1, 3)$` , должна делать то же самое, что, очевидно, неэффективно. Здравомыслящий программист попытался бы найти средства сохранить вычисленные в ходе первой итерации элементы списка C , с тем чтобы можно было получить к ним прямой доступ во время второй итерации.

Для того чтобы образовать такое поведение при помощи порождения лемм, мы можем воспользоваться простой схемой аннотации: каждый вызов, решение которого требуется для порождения соответствующего факта, помечается символом Δ . (Этот символ направлен острием вверх, поскольку порождение лемм является характерной особенностью решения задач методом снизу вверх; добавление символа Δ к стандартной стратегии дает вид поведения, в котором смешиваются методы сверху вниз и снизу вверх.)

В результате применения этой схемы к нашему примеру будет модифицирована итеративная процедура `печать*`, в которую просто вставляется символ Δ ; она будет выглядеть теперь следующим образом

```
печать*( $z, i, j$ ) если  $i \leq j$ ,  $\Delta \exists(w, i, z)$ ,
    печать( $w$ ), печать*( $z, i + 1, j$ )
```

Все другие процедуры программы останутся без изменений. Итогом этой модификации является то, что в ходе решения вызова `печать*($C, 1, 3$)` каждый активированный помеченный вызов $\Delta \exists(w, i, C)$ при $i = 1, 2, 3$ будет порождать новый факт, как только будет успешно вычислено значение переменной w . Таким образом, на данном этапе исполнения к программе добавятся факты

```
∃(10, 1, C)
∃(20, 2, C)
∃(30, 3, C)
```

Когда будет решаться последний вызов `печать** $(C, 1, 3)$` ; из целевого утверждения вызовы `∃`, которые он порождает для нахождения элементов списка C , могут быть решены непосредственно с помощью этих новых фактов, а не путем обращения к исходным процедурам `∃`. Для того чтобы обеспечить именно такой ход событий, требуется принять некоторые соглашения относительно приоритета вызова входных процедур и порожденных фактов. Можно было бы ввести такое простое правило, согласно которому вызовы всегда пытаются обращаться сначала к «самым молодым» процедурам, т. е. к тем процедурам, которые появились в ходе исполнения программы самыми последними. В рассматриваемом примере это правило, очевидно, работает

удовлетворительно, однако при других обстоятельствах могло бы потребоваться более сложное соглашение.

При нормальном ходе событий интерпретатор с целью поиска альтернативных решений после нахождения первого решения вызова **печать**** (*C, I, 3*) стал бы в конце концов выполнять процесс возврата, и поэтому он обнаружил бы, что в каждой точке ветвления, где уже выбирались новые факты *з*, исходные процедуры *з* остались до сих пор неиспользованными. Для того чтобы избавиться от неэффективного и нежелательного второго решения вызова **печать**** (*C, I, 3*), мы можем обрезать ветви, выходящие из этих точек ветвления, следующим образом используя оператор отсечения в целевом утверждении:

? **длина**(*C, n*), **печать***(*C, I, n*), **печать****(*C, I, n*), /

Реализации, в которых каким бы то ни было методом в процессе исполнения программ порождаются новые факты, должны также обладать способностью решать в период исполнения, как хранить эти факты и как сделать их доступными в дальнейшем. Это может означать, что программное обеспечение, обычно используемое для организации индексирования в период компиляции, должно быть также доступным и во время исполнения, в результате чего в процессе исполнения возникнут некоторые дополнительные расходы.

Модифицированная программа 26 страдает незначительным дефектом стиля, заключающимся в помещении символа Δ внутрь множества процедур **печать***, поскольку эти процедуры теперь уже нельзя вызывать, не порождая при этом новые факты, что приводит к некоторой потере их гибкости. Более изящный подход, дающий более модульную структуру, состоит в использовании нового множества процедур, цель которого — только образовывать множество фактов, не делая никаких предположений относительно их дальнейшего употребления. Это новое множество процедур, которое можно также назвать модулем, генерирующим списки, могло бы выглядеть следующим образом:

породить(*z*) если Δ **длина**(*z, n*), **элементы**(*z, I, n*)
элементы(*z, i, j*) если $i > j$
элементы(*z, i, j*) если $i \leq j$, Δ **э**(*w, I, z*),
элементы(*z, i + 1, j*)

Целевое утверждение программы запишется тогда в виде

? **породить**(*C*), **длина**(*C, n*), **печать***(*C, I, n*), **печать****(*C, I, n*), /

и, таким образом, его первый вызов порождает полное представление списка *C* посредством фактов. Оставшаяся часть программы будет в точности такая же, как и данная первоначально

в программе 26, т. е. без символов Δ . После того как список C будет порожден в результате вычисления его длины и элементов из имеющихся данных для списков A и B , оба задания, связанные с печатью, могут выполняться посредством прямой выборки элементов C из порожденных фактов. В итоге мы имеем превосходный алгоритм и написанную в хорошем стиле программу с чистой семантикой.

IV.4.4. Имена для структур данных, представленных в виде фактов

Программа 26 вычисляла и печатала сумму конкретных векторов $C = A \oplus B$. Это допущение было встроено в процедуры длина и э. В более общем случае нам может потребоваться программа, в целевом утверждении которой указывается подлежащая вычислению и распечатке произвольная сумма векторов, и в связи с этим возникает проблема обобщения двух упомянутых процедур. На интуитивном уровне мы могли бы представить их в новой форме следующим образом:

$$\begin{aligned} \text{длина}(?, n) &\text{ если } \text{длина}(x, n), \text{длина}(y, n) \\ \text{э}(w, i, ?) &\text{ если } \text{э}(u, i, x), \text{э}(v, i, y), \text{сложить}(u, v, w) \end{aligned}$$

Здесь позиции аргументов, отмеченные знаком ?, должны быть каким-либо образом приспособлены для помещения на них имени векторной суммы $x \oplus y$, причем x и y могут быть любыми списками, которые передаются обращающимися к этим процедурам вызовами. На указанные позиции нельзя просто поставить некоторую произвольно выбранную третью переменную, скажем z , поскольку в этом случае процедуры перестали бы являться логически правильными утверждениями о списках. Более того, на позициях, отмеченных знаком ?, должны содержаться ссылки и на x , и на y , так чтобы выбранные списки могли стать известными нашим процедурам.

Один из способов получения желаемого результата заключается в использовании структурированного терма *сумма*(x, y), который может служить именем для списка $x \oplus y$ и заменять его на каждой позиции, отмеченной знаком ?. В целевом утверждении программы можно теперь указывать какие угодно конкретные списки, употребляя для них с этой целью структурированные имена, такие как *сумма*(A, B) или *сумма*($A, \text{сумма}(A, B)$). Полная программа, предназначенная для вычисления и распечатки суммы векторов $A \oplus B$, в которой используются обобщенные процедуры и порождение лемм, имела бы тогда следующую структуру.

Программа 27

```
? z = сумма(A,B), породить(z), длина(z,n),
    печать*(z,I,n), печать***(z,I,n),/
    породить(z) если Δ длина(z,n), элементы(z,I,n)
    элементы(z,i,j) если i > j
    элементы(z,i,j) если i ≤ j, Δ э(w,i,z),
        элементы(z,i+1,j)
    длина(сумма(x,y),n) если длина(x,n), длина(y,n)
    э(w,i,сумма(x,y)) если э(u,i,x), э(v,i,y),
        сложить(u,v,w)
```

а также процедуры печать* и печать** из программы 26 и факты длина и э, представляющие списки A и B

Следует отметить пользу, которую приносит употребление первого вызова **=** в целевом утверждении: он избавляет нас от необходимости указывать имя **сумма(A,B)** в различных других местах этого целевого утверждения.

Приведенная программа дает хорошее поведение в период исполнения. Сначала вызывается модуль генерирования списков, как это описано в предыдущем разделе, для того, чтобы образовать представление списка $A \oplus B$ в виде фактов; каждый факт в этом представлении будет иметь имя **сумма(A,B)**. Затем при выполнении последующих заданий, связанных с печатью, осуществляется прямой доступ к данным. Если нам потребуется обработать какие-то другие суммы векторов, то для этого необходимо только поменять имя, указанное в первом вызове целевого утверждения, и задать соответствующие факты длина и э, которые представляют выбранные входные списки.

Рассматриваемый метод именования может стать утомительным, когда мы имеем дело с очень сложными составными именами. Допустим к примеру, что входные данные состоят из списков A_1, A_2, B_1 и B_2 , и нам нужно вычислить и распечатать список $(A_1 \oplus A_2) \oplus (B_1 \oplus B_2)$. Соответствующее ему имя **сумма(сумма(A1,A2), сумма(B1,B2))** слишком громоздкое для написания, и, кроме того, его употребление могло бы привести к значительным затратам, связанным с хранением данных и выполнением унификации в период исполнения. В этом случае можно использовать другой возможный стиль программирования, при котором мы обходимся без структурированных имен и который делает программы более понятными. Новый стиль связан с модификацией приведенных в программе 27 обобщенных процедур длина и э. Теперь они примут вид

```
длина(z,n) если сумма(x,y,z), длина(x,n), длина(y,n)
э(w,i,z) если сумма(x,y,z), э(u,i,x), э(v,i,y),
    сложить(u,v,w)
```

Здесь предикат сумма (x, y, z) означает, что $z = x \oplus y$. Искомому выходному списку можно дать простое имя, скажем C , и следующим образом указать его в целевом утверждении

? породить(C), длина(C, n), печать*(C, I, n), печать**(C, I, n), /

Теперь остается только сообщить программе о том, что $C = (A1 \oplus A2) \oplus (B1 \oplus B2)$. Это легко можно сформулировать, добавив к утверждениям программы еще три факта

сумма($A1, A2, A3$)
сумма($B1, B2, B3$)
сумма($A3, B3, C$)

Здесь $A3$ и $B3$ являются именами промежуточных сумм $A1 \oplus A2$ и $B1 \oplus B2$ соответственно. За исключением указанных изменений все процедуры программы останутся в точности такими же, как и в программе 27, а база данных содержит теперь представления в виде фактов списков $A1, A2, B1$ и $B2$. В ходе исполнения для порождения списка C вызывается генератор списков, в результате чего с помощью процедур длина и э запрашивается содержимое списков $A3$ и $B3$. Поскольку непосредственно они в базе данных не содержатся, далее осуществляются рекурсивные вызовы процедур длина и э для того, чтобы выяснить содержимое составных частей $A3$ и $B3$ — списков $A1, A2$ и $B1, B2$ соответственно. Как только эти входные списки будут выбраны из базы данных, могут выполняться различные процессы сложения и, следовательно, будет вычислено содержимое списка C , представлено в виде фактов и, наконец, распечатано. Исполнение программы оказывается весьма эффективным, причем выбор входных и выходных данных в ней легко может быть изменен.

IV.4.5. Еще раз о задаче нахождения собственных значений и собственных векторов

Обсуждавшиеся в предыдущем разделе методы именования являются адекватными только для тех задач, которые содержат заранее известное конечное число структур данных, представленных в виде фактов. Для написания программ, в ходе исполнения которых порождаются последовательности структур данных, причем длины этих последовательностей изначально неизвестны, требуются несколько более сложные методы. Подобная ситуация обычно возникает в программах, связанных с итерацией матриц: программа завершает работу только при выполнении некоторого вида условия сходимости.

Описанная в гл. III задача нахождения собственных значений и собственных векторов дает как раз такой пример. Исполь-

зованный там алгоритм ее решения, начиная с заданного исходного вектора V и фиксированной матрицы M , порождает последовательность дальнейших векторов $M.V, M^2.V, M^3.V$ и т. д. до тех пор, пока один из них не будет удовлетворять некоторым условиям точности приближения; этот вектор и является тогда искомым решением.

Если каждый вектор в последовательности должен быть представлен посредством фактов (а не в виде структурированного терма, как это имело место в программе, приведенной в гл. III), то ему следует дать имя, отличающее его от всех остальных векторов. Это значит, что алгоритм помимо последовательности векторов будет порождать также последовательность их имен. Для обеспечения эффективности важно, чтобы эти имена в процессе построения последовательности не увеличивались в размерах. Так, например, было бы непрактично для векторов $M.V, M^2.V, \dots$ и т. д. использовать имена вида $\text{умнож}(M, V)$, $\text{умнож}(M, \text{умнож}(M, V))$, ... и т. д., поскольку их хранение и обработка снизили бы эффективность.

Один из способов достижения удовлетворительной схемы именования заключается в том, чтобы включать в каждое имя число, показывающее, сколько раз необходимо выполнить умножение на матрицу M для порождения именуемого вектора из начального вектора V . Таким образом, мы могли бы использовать имена

<i>вектор</i> ($M, V, 0$)	для вектора V
<i>вектор</i> ($M, V, 1$)	для вектора $M.V$
<i>вектор</i> ($M, V, 2$)	для вектора $M^2.V$
и т.д.	

Эти структурированные термы дают бесконечную последовательность различных имен одинаковой длины. Исходную программу 10 для решения задачи нахождения доминантного собственного вектора матрицы можно преобразовать теперь следующим образом.

Программа 28

```

? вывести(в, вектор( $M, V, 0$ )), точн( $M, v$ )
вывести(в, в)
вывести(в, z) если умнож( $M, z, z'$ ),
                    вывести(в, z')
умнож( $M, z, z'$ ) если z = вектор( $M, V, k$ ),
                    z' = вектор( $M, V, k + 1$ ),
                    породить(z')

```

а также

(а) процедуры точн, кодирующие проверку точности приближения;

- (b) факты длина и э, представляющие входной вектор V при помощи имени *вектор*(M , V , 0);
- (c) некоторое подходящее представление структуры данных для матрицы M ;
- (d) процедуры длина и э, способные для любого $k \geq 0$ вычислять длину и элементы вектора *вектор*(M , V , $k+1$), полученного в результате умножения матрицы M на *вектор*(M , V , k); именно в этих процедурах кодируются детали операции умножения матриц, а их логика будет включать поиск элементов M в представлении, определенном в (c);
- (e) использованный в программе 27 модуль генерирования списков, в котором применяется порождение лемм с целью получения представления в виде фактов каждого нового вектора для того, чтобы подготовиться к следующему шагу итерации.

Вот как работает эта программа. Первый вызов процедуры *вывести* присвоит переменной v значение *вектор*(M , V , 0), а последующий вызов точн проверит входной вектор на точность приближения. Если этот вектор не удовлетворяет требуемой точности, то произойдет возврат и будет вызвана вторая процедура *вывести*, в результате чего вызов *умнож* использует текущее имя *вектор*(M , V , 0) для того, чтобы построить следующее имя *вектор*(M , V , 1). Вызов *породить*, входящий в тело процедуры *умнож*, образует представление в виде фактов нового вектора с этим именем. Затем еще раз будет активирован вызов точн и на точность приближения проверится вектор $v := \text{вектор}(M, V, 1)$. Весь описанный процесс построения очередного имени, образования соответствующего ему вектора и проверки его на точность приближения будет неоднократно повторяться до тех пор, пока не будет достигнута требуемая сходимость.

Это поведение, судя по времени исполнения программы, оказывается эффективным, однако память в нем эффективно использоваться не будет, если не предусмотреть меры, обеспечивающие запись каждого нового вектора на месте его предшественника. Необходимость в деструктивном присваивании возникает главным образом в алгоритмах, образующих бесконечные последовательности структур данных. В большинстве реализаций предусмотрено некоторое количество автоматических средств экономии памяти, причем часть из них может отвечать также на соответствующие директивы программиста. Общая проблема экономии памяти в период исполнения связана с целым рядом теоретических и практических трудностей и полностью еще не решена. В различных реализациях используются свои собственные механизмы управления распределением памяти. Таким образом, программист, желающий извлечь наибольшую выгоду из конкретного представления данных, должен

хорошо понимать имеющуюся у него реализацию и программировать в соответствии с ней.

Следует, наконец, отметить, что на самом деле в рассматриваемом примере не обязательно включать V и M в имена векторов. Фактически эти имена можно в значительной степени упростить, используя вместо них лишь символы $0, 1, 2, \dots$ и т. д., в результате чего программу станет легче читать и она будет чуть более эффективной. Несколько первых утверждений из программы 28 можно было бы в этом случае представить в следующем упрощенном и сжатом виде:

```
? вывести(v, 0), точн(M, v)
вывести(v, v)
вывести(v, k) если породить(k + 1), вывести(v, k + 1)
а также указанные выше составные части (a), ..., (e),
приспособленные к упрощенным именам
```

Так, составляющая (b) обеспечивает входные данные вида

$\text{э}(1, 1, 0)$	$\text{э}(1, 2, 0)$	представляющие вектор $V = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
$\text{длина}(0, 2)$		

а итерация порождает такие представления векторов

$\text{э}(5, 1, 1)$	$\text{э}(29, 1, 2)$
$\text{э}(7, 2, 1)$	$\text{э}(43, 2, 2)$
$\text{длина}(1, 2)$	$\text{длина}(2, 2)$... и т. д.

Необходимость в более сложных именах, включающих и другие символы, подобные M и V , возникает только в тех программах, где требуется различать несколько входных векторов и несколько матриц.

IV.5. Исторический очерк

Первое сравнительное исследование структур данных для логических программ появилось в статье Ковалевского (1974а). В ней даются интересные примеры использования как термов, так и фактов для представления данных в задачах синтаксического анализа, сортировки и составления планов.

Представления данных исследовались также в статье Ковалевского (1979б) о логических алгоритмах. В ней содержится новый пример, в котором каждая отдельная компонента структуры данных представляется посредством импликации, а не простого факта, что дает более тонкий способ квазиэкстенсионального определения отношений. В этой статье вводятся также некоторые простые аннотации к программам, которые специфици-

рут управляемое смещивание стратегий исполнения методами сверху вниз и снизу вверх и которые можно использовать для извлечения механизма порождения лемм, описанного ранее в данной главе.

Приведенные здесь программы для задачи о палиндроме взяты из более полного исследования логических структур данных, включенного в диссертацию Хоггера (1979а). Задача логического преобразования программы, основанной на представлении посредством термов, в программу, основанную на представлении посредством фактов (например, преобразование программы 21 в программу 22), иллюстрируется как в упомянутой работе, так и в статье (Хоггер, 1981). Примеры различных видов термов, предназначенных для обработки списков, деревьев и других структур, были также с достаточной полнотой продемонстрированы Кларком (1979) и Кларком и Тернлундом (1977).

Подробное изложение обычной организации внутреннего представления термов в реализациях дается Уорреном (1977а), а схемы индексирования для хранения и выборки фактов и других утверждений описываются в статье Кларка и Маккейба (1980). Обе эти темы рассматриваются Уорреном и др. (1977) при сравнении Пролога с Лиспом. Ссылки на работы, посвященные специальной теме логических баз данных, можно найти в гл. VIII.

V. Верификация программ

Пользуемся мы логикой или любой другой формальной системой программирования, всегда желательно иметь какие-либо практические средства, позволяющие решать, является ли каждая конкретная программа правильной. При широком понимании программа считается правильной, если в результате ее исполнения получаются правильные решения рассматриваемой задачи. Правильность вычислений решений связана с некоторой спецификацией задачи, представленной независимо от программы. Разумеется, предполагается также, что интерпретатор программы сам правильно исполняется на машине.

В контексте логического программирования проблема верификации затрагивает ряд вопросов. Возможно, наиболее важным из них является вопрос о правильности вычисленных решений. Другой вопрос: все ли правильные решения задачи действительно вычисляются исходя из программы? Верификация — это процесс доказательства того, что программа удовлетворяет или не удовлетворяет требованиям такого рода. Подобное исследование было бы не нужным, если бы все практически используемые логические программы могли сами по себе считаться спецификациями. Можно утверждать, конечно, что некоторые программы действительно содержат формулировки, которые стоило бы вообще рассматривать как очевидно правильные. Чаще случается, однако, что для построения эффективной нетривиальной программы требуется составлять ее из утверждений, правильность которых с первого взгляда не очевидна.

Как можно было бы ожидать, содержание этой главы носит более теоретический характер, чем в предыдущих главах. В ней приводится ряд формальных определений наиболее важных свойств программ и дается общая схема одного из подходов к практической верификации.

V. 1. Вычисляемые и специфицируемые отношения

Чтобы formalизовать такие понятия, как правильность, необходимо прежде всего иметь точные способы характеристизации того, что программа вычисляет, а спецификация определяет. Для

этого в свою очередь требуется принять некоторые соглашения относительно класса рассматриваемых программ, а такжевести подходящую терминологию для обсуждения их свойств.

V.1.1. Обозначения и терминология

Для целей верификации удобно потребовать выполнения следующего условия: интересующая нас программа должна быть устроена таким образом, чтобы ее целевое утверждение G содержало только один вызов, т. е. имело вид

$$G : ? g(t)$$

где t — некоторый кортеж из n термов. Множество всех процедур, используемых в программе, обозначим через P . Будем считать, что оно включает в себя все встроенные процедуры, на которые имеются ссылки в программе. Всю программу можно представить поэтому парой (P, G) .

Спецификацией программы является любое множество определений (не обязательно записанных в логических терминах), которое точно задает содержимое каждого из отношений, встречающихся в программе. В частности, спецификация будет определять содержимое отношения, указанного в целевом утверждении G . В дальнейшем это отношение называется *основным специфицируемым отношением* и обозначается, как правило, символом R^* . Если, к примеру, целевым утверждением программы является

$$G : ? \text{подмнож}(w : A : \emptyset, A : B : C : \emptyset)$$

то основное специфицируемое отношение носит имя **подмнож**, так что мы можем написать $R^* = \text{подмнож}$. Символ S обычно используется для обозначения спецификации. Спецификация программы, имеющей приведенное выше целевое утверждение, будет точно определять, какие кортежи из двух элементов принадлежат отношению **подмнож**, а какие — нет, и с точки зрения программиста спецификация будет считаться надежным определением этого отношения. В частности, S будет определять, какие двухэлементные кортежи являются правильными решениями целевого утверждения G .

Пусть каждый предикат **подмнож** (x, y) означает, что множество x есть подмножество множества y , и пусть множества представляются с помощью функтора $:$ и константы \emptyset , где \emptyset обозначает пустое множество, а терм вида $v : z$ обозначает множество, получаемое объединением множества $\{v\}$ с некоторым другим множеством z (здесь v — произвольный элемент). Тогда в указанном выше целевом утверждении G спрашивается, для каких элементов w множество $\{w, A\}$ является подмножеством

$\{A, B, C\}$. Заметим, что это *ограниченный* запрос об отношении подмнож, поскольку G фокусируется только на определенных двухэлементных кортежах, принадлежащих этому отношению. Более формально, мы говорим, что в данном примере целевое утверждение G *выделяет* некоторое подотношение (*интервал*) отношения R*, а именно подотношение

$$\{(A : A : \emptyset, A : B : C : \emptyset), \\ (B : A : \emptyset, A : B : C : \emptyset), \\ (C : A : \emptyset, A : B : C : \emptyset)\}$$

Остальные двухэлементные кортежи из R*, такие как $(B : \emptyset, B : D : \emptyset)$ целевым утверждением G, очевидно, не запрашиваются и поэтому не принадлежат выделенному G интервалу.

Некоторые программы могут иметь целевые утверждения, выделяющие все отношение R*. Чтобы добиться этого, можно сформулировать *наиболее общее целевое утверждение* G*, все параметры которого являются различными переменными, например:

$$G : ? \text{ подмнож}(x, y)$$

Этот способ не единственный. Если, к примеру, R* — некоторое другое отношение, скажем, $R^* = \{(A, B), (C, B)\}$ и

$$G : ? R^*(x, B)$$

то целевое утверждение G выделяет все отношение R*, но не является наиболее общим.

Интервал, выделяемый каждым целевым утверждением

$$G : ? R^*(t)$$

есть множество решений G, соответствующих спецификации S, причем каждое решение является некоторым подстановочным примером $t\Theta$ кортежа термов t. Итак, формально, интервал целевого утверждения G относительно спецификации S есть множество

$$\{t\Theta | t\Theta \in R^*\}$$

где S специфицирует R*. Заметим, что если этот интервал пуст, то в соответствии со спецификацией S целевое утверждение G не должно иметь решений. В общем случае решения, разумеется, существуют; чтобы подчеркнуть соответствие решений спецификации S, каждое такое решение $t\Theta$ назовем *специфицируемым решением* G. Является ли оно также решением, вычисляемым программой, — совсем другой вопрос, и именно на этот вопрос должна дать ответ верификация программы.

V.1.2. Вычисляемое отношение

Вычисляемое отношение программы (P, G) — это множество всех тех решений целевого утверждения G , которые вычисляются исходя из программы с помощью заданного множества процедур P . Это отношение обозначается, как правило, через R , а решения из R называются *вычисляемыми решениями*.

Предположим, что программа допускает несколько успешных вычислений. Каждое из них дает некоторое выходное присваивание Θ термов переменным из исходного целевого кортежа t . Таким образом устанавливается, что $t\Theta$ решает G в соответствии с P . Если кортеж $t\Theta$ не содержит переменных, то он принадлежит R ; в противном случае этот кортеж можно рассматривать как общее обозначение для всех его подстановочных примеров, не содержащих переменных, причем каждый из них принадлежит R . Заметим, что если отношение пусто, то программа неразрешима.

Из гл. I и II мы уже знаем фундаментальное свойство логического программирования, заключающееся в том, что решение $t\Theta$ целевого утверждения

$$G : ?R^*(t)$$

вычислимо с помощью P (в предположении резолютивного исполнения программы) тогда и только тогда, когда $P \models R^*(t\Theta)$. Это дает нам возможность следующим образом формально определить вычисляемое отношение R .

Определение 1.

$$R = \{t\Theta \mid P \models R^*(t\Theta)\}$$

где P — множество процедур, а $?R^*(t)$ — целевое утверждение.

Конкретный пример поможет внести ясность в эти понятия. Пусть у нас имеется следующая программа:

Программа 29

$$\begin{aligned} G : & ? \text{подмнож}(\omega : A : \emptyset, A : B : C : \emptyset) \\ & \text{подмнож}(x, y) \text{ если } \text{пустое}(x) \\ & \text{подмнож}(v : x', y) \text{ если } v \in y, \\ & \quad \text{подмнож}(x', y) \\ & v \in v : z \\ & v \in u : z \text{ если } v \in z \end{aligned}$$

Здесь свойство *пустое*(x) имеет место тогда, когда x — пустое множество (не представимое в виде $v : x'$). Множества процедур для предикатов *подмнож*, \in и *пустое* (в последнем случае процедуры предполагаются встроенными) все вместе составляют P . Программист рассчитывает, что эта программа будет исследовать отношение вложения в связи с определенными множест-

вами. Имеющееся в виду содержимое отиошения $R^* =$ подмнож определяется некоторой независимой спецификацией S . Программа имеет три успешных вычисления, которые присваивают исходной целевой переменной w значения A , B и C . Поэтому каждое i -е вычисление ($i = 1, 2, 3$) добавит к вычисляемому отиошению R вычисляемое решение ($w : A : \emptyset$, $A : B :: C : \emptyset$) Θ_i , где Θ_i — одна из подстановок $\{w := A\}$, $\{w := B\}$ или $\{w := C\}$. Следовательно, содержимое отиошения R таково:

$$\begin{aligned} R = & \{(A : A : \emptyset, A : B : C : \emptyset), \\ & (B : A : \emptyset, A : B : C : \emptyset), \\ & (C : A : \emptyset, A : B : C : \emptyset)\}. \end{aligned}$$

Заметим, что в соответствии с определением 1 для каждого решения $t\Theta$ из R целевое утверждение

? подмнож($t\Theta$)

разрешимо с помощью множества процедур P . Это происходит потому, что ? подмнож ($t\Theta$) решается с помощью P тогда и только тогда, когда целевое утверждение ? подмнож (t) решается с выходимым присваиванием Θ , что в свою очередь является просто следствием недетерминизма логических процедур из P по отиошению к входу и выходу.

В только что рассмотренном примере вычисляемое отиошение R оказывается интервалом целевого утверждения G относительно приведенной в предыдущем разделе спецификации S . Содержательно это как раз и означает, что программа вычисляет в точности то, что требуется согласно предполагаемой спецификации. Таким образом, множества специфицируемых решений и вычисляемых решений совпадают.

V. 2. Правильность программ: определения

Полная правильность логической программы предполагает выполнение двух необходимых условий. Во-первых, каждое вычисляемое программой решение должно быть правильным относительно данной спецификации; это свойство называется *частичной правильностью* программы. Во-вторых, каждое решение, присываемое спецификацией целевому утверждению, должно быть вычисляемым с помощью программы; это свойство называется *полнотой* программы. Если выполняются оба этих свойства, то говорят, что программа *полностью правильна*¹⁾. Для точ-

¹⁾ Встречается также другой вариант перевода терминов *partial correctness* (частичная правильность) и *total correctness* (полная правильность) — частичная корректность и тотальная корректность соответственно. — Прим. перев.

ногого определения данных понятий удобно считать, что сама спецификация S записана на языке логики предикатов первого порядка. Это допущение дает возможность следующим образом представить основное специфицируемое отношение R^* .

Определение 2

$$R^* = \{T \mid S \models R^*(T)\}$$

где T — произвольный кортеж из n термов (специфицируемое решение), а $G : ? R^*(t)$ — целевое утверждение программы. Кроме того, мы можем теперь определить также интервал целевого утверждения G (множество правильных решений G) относительно спецификации S .

Определение 3. Интервал целевого утверждения G относительно спецификации S есть множество

$$\{t\Theta \mid S \models R^*(t\Theta)\}$$

Таким образом, определения 1, 2 и 3 описывают вычисляемые и специфицируемые решения в терминах логического следования из множеств R и S . Значение этих определений станет ясным из дальнейшего.

V.2.1. Частичная правильность

Программа (P, G) частично правильна относительно спецификации S тогда и только тогда, когда она удовлетворяет следующему требованию: для всех подстановок Θ всякое вычисляемое решение $t\Theta$ должно быть также и специфицируемым решением. В приводимой ниже формулировке частичная правильность определяется тремя эквивалентными способами.

Определение 4. Частичная правильность $(P, G : ? R^*(t))$ относительно S .

Программа $(P, G : ? R^*(t))$ частично правильна относительно спецификации S тогда и только тогда, когда выполняется одно из следующих условий:

- (а) для всех Θ , если $t\Theta \in R$, то $t\Theta \in R^*$;
- (б) $R \sqsubseteq$ (интервал G относительно S);
- (с) для всех Θ , если $P \models R^*(t\Theta)$, то $S \models R^*(t\Theta)$.

Рассмотрим в качестве примера программу 29. Было показано, что у этой программы имеется три различных вычисляемых решения вида $t\Theta$, где $t = (\omega : A : \emptyset, A : B : C : \emptyset)$, а Θ — некоторое присваивание переменной ω . Для того чтобы программа 29 была частично правильной, каждый кортеж термов $t\Theta$ должен иметь вид $(множ_1, множ_2)$, причем в соответствии с задаваемой S спецификацией отношения **подмнож** множество *множ_1*

является подмножеством множ-2. Например, в случае, когда $\Theta = \{w := B\}$, мы имеем вычисляемое решение $t\Theta = (B : A : \emptyset, A : B : C : \emptyset)$. Множество $\{B, A\}$ есть подмножество множества $\{A, B, C\}$, и потому спецификация S , по всей видимости, будет определять, что пара $(B : A : \emptyset, A : B : C : \emptyset)$ принадлежит отношению подмнож. В этом случае $t\Theta$ является также одним из специфицируемых (т. е. правильных) решений целевого утверждения. То же самое справедливо и для двух других вычисляемых решений, и, стало быть, программа 29 является частично правильной.

Рассматриваемое понятие частичной правильности логических программ по своей сути подобно тому, которое используется в доказательстве правильности традиционных программ. И в том и другом контексте частичная правильность сама по себе не требует, чтобы каждое решение действительно вычислялось в результате исполнения программы; требуется только то, что уж если решение вычисляется, то оно должно удовлетворять задаваемому спецификацией отношению между входом и выходом.

Заметим также, что все неразрешимые программы, согласно определению 4, являются частично правильными. С первого взгляда это может показаться довольно странным, поскольку отсюда следует, например, что процедуры программы могут быть бессмысленными по отношению к проблемной области утверждениями, не способными выдать никаких решений целевого утверждения, и тем не менее программа считается частично правильной. Эта очевидная терминологическая аномалия становится приемлемой, если признать, что неразрешимая программа конечно же *не является неправильной* — программа, не вычисляющая вообще никаких решений, не может вычислить и неправильного решения. Сказать, что логическая программа частично правильна — это значит сказать только то, что она не вычисляет решений, которые противоречили бы спецификации.

Оказывается действительно важным, чтобы наши определения были достаточно гибкими и позволяли неразрешимым программам быть частично правильными, поскольку такие программы образуют ценную часть репертуара программиста. Можно, например, взять в качестве входных данных список L , представленный обычным образом с помощью некоторого множества P фактов \mathbf{e} , и потребовать демонстрации того, что некоторый элемент A не имеет вхождений в L ни на какой позиции i . Простейший способ сделать это — сформулировать целевое утверждение

$$G : ? \mathbf{e}(A, i, L)$$

и убедиться в том, что в результате исполнения программы (P, G) не будет вычислено никаких решений. Хотя эта програм-

ма неразрешима, тем не менее она выполняет в точности то, что требуется, и, следовательно, нет оснований отрицать ее право называться «частично правильной».

V.2.2. Полнота

Программа (P, G) является полной относительно спецификации S тогда и только тогда, когда она удовлетворяет следующему требованию: для любой подстановки Θ всякое специфицируемое решение $t\Theta$ должно быть также и вычисляемым решением. В приводимой ниже формулировке полнота определяется тремя эквивалентными способами.

Определение 5. Полнота $(P, G : ? R^*(t))$ относительно S .

Программа $(P, G : ? R^*(t))$ полна относительно спецификации S тогда и только тогда, когда выполняется одно из следующих условий:

- (a) для всех Θ , если $t\Theta \in R^*$, то $t\Theta \in R$
- (b) (интервал G относительно S) $\subseteq R$
- (c) для всех Θ , если $S \models R^*(t\Theta)$, то $P \models R^*(t\Theta)$

Можно выразить это по-другому: вместо того, чтобы говорить о полноте программы (P, G) , можно сказать, что множество процедур P является полным для G , или, более компактно, что P является G -полным. Это как раз и означает, что процедур из P достаточно для вычисления всех решений из выделяемого целевым утверждением G интервала. Если же множество процедур P не является G -полным, то это значит, что P содержит недостаточно информации о специфицируемом отношении R^* для того, чтобы сделать все правильные решения G вычисляемыми. Рассмотрим еще раз программу 29. В ней множество процедур P , конечно же, G -полно, поскольку каждое из трех специфицируемых решений целевого утверждения G является вычислимым с помощью P . Допустим теперь, что вторую процедуру ϵ мы несколько видоизменили:

$$v \in A : z \text{ если } v \in z$$

и получили в результате новое множество процедур P' . В этом случае, оказывается, множество P' не является G -полным, потому что третье правильное решение $(C : A : \emptyset, A : B : C : \emptyset)$ целевого утверждения G уже не вычислимо. Модификация процедуры ϵ ограничивает ее область действия только множествами вида $A : z$, что ведет к потере информации о принадлежности в других множествах, а значит и к потере информации об отношении подмнож. С другой стороны, P' является полным для

ряда других целевых утверждений, таких как

$$G' : ? \text{подмнож}(w : A : \emptyset, A : B : \emptyset)$$

Некоторые множества процедур полны независимо от того, какое поставлено целевое утверждение G . В качестве примера можно взять множество процедур P из программы 29. Какой бы вызов процедуры **подмнож** мы ни сформулировали в целевом утверждении, все специфицируемые решения этого утверждения будут вычислиммы с помощью P . Когда множество процедур P обладает таким свойством, мы говорим просто, что оно является **полным**. Формальное определение полноты таково.

Определение 6. Полнота P относительно S .

Множество процедур P является полным относительно спецификации S тогда и только тогда, когда

$$\text{для всех } T, \text{ если } S \models R^*(T), \text{ то } P \models R^*(T)$$

Следует отметить, что в этом определении для обозначения кортежей термов используется общий символ T , т. е. они не обязаны иметь вид $t\Theta$, где t — кортеж термов из целевого утверждения. Это связано с тем, что полное множество процедур может иметь дело со всеми возможными целевыми кортежами термов и, следовательно, со всеми возможными выборами входных и выходных аргументов.

Наконец, заметим, что если множество процедур P является полным, то оно с необходимостью является G^* -полным, т. е. полным для наиболее общего целевого утверждения

$$G^* : ? R^*(t)$$

в котором t — кортеж различных переменных. Выделяемый этим целевым утверждением интервал охватывает все специфицируемое отношение R^* , и полное множество процедур P содержит достаточно информации относительно R^* , чтобы сделать все его элементы вычисляемыми решениями G^* . Это значит, в частности, что если мы сформулируем целевое утверждение

$$G^* : ? \text{подмнож}(x, y)$$

и будем решать его с помощью полного множества процедур из программы 29, то в качестве вычисленных выходных данных мы должны будем получить всю бесконечную совокупность пар множеств, удовлетворяющих отношению **подмнож**.

V.2.3. Полная правильность

Программа (P, G) называется полностью правильной относительно спецификации S тогда и только тогда, когда она является и частично правильной, и полной. В приводимой ниже

формулировке полная правильность определяется тремя эквивалентными способами.

Определение 7. Полная правильность ($P, G : ? R^*(t)$) относительно S .

Программа ($P, G : ? R^*(t)$) является полностью правильной относительно спецификации S в том и только том случае, когда выполняется одно из следующих трех условий:

- (a) для всех $\Theta \models \Theta \in R$ тогда и только тогда, когда $\Theta \models R^*$
- (b) (интервал G относительно S) = R
- (c) для всех $\Theta \models P = R^*(\Theta)$ тогда и только тогда, когда $S \models R^*(\Theta)$

На менее формальном уровне программа полностью правильна относительно S , когда вычисляемые ею решения в точности совпадают с решениями, специфицируемыми S . Программа 29 дает пример полностью правильной программы. Если программа не является полностью правильной, то либо она вычисляет некоторое неправильное решение, либо оказывается не в состоянии вычислить какое-то правильное; первый случай означает, что ее процедуры представляют собой неверные утверждения о проблемной области, а второй — что они содержат недостаточно информации о ней. При нормальных условиях первый случай более серьезен, хотя и второй может также быть важным, когда в программе содержатся отрицательные вызовы, и их решение основывается на методе отрицание как неудача.

V.3. Правильность программ: достаточные условия

Формальные определения 4, 5 и 7 устанавливают критерии правильности программы, в основе которых лежит понятие вычислимости и которые не накладывают никаких требований на ее отдельные процедуры. Принимая это во внимание, мы рассмотрим следующую программу.

Программа 30

```

G : ? умножить(4, 6, z)
P1 :   умножить(x, 3, x)
P2 :   умножить(w, 2, 24)
P3 :   умножить(x, 6, z) если умножить(x, 3, w),
          умножить(w, 2, z)

```

Здесь предикат $\text{умножить}(x, y, z)$ предназначается, как обычно, для представления арифметического умножения $z = x * y$. Допустим, что имеется некоторая спецификация S , формально со-поставляющая предикату $\text{умножить}(x, y, z)$ это имеющееся в виду значение.

В целевом утверждении G требуется найти значение z произведения $4 * 6$. Согласно спецификации S должно существовать только одно специфицируемое решение $z := 24$. Программа вычисляет тоже одно решение, и им действительно является $z := 24$. Стало быть, программа полностью правильна.

Рассмотрим теперь ее отдельные процедуры. Р3 утверждает, что для вычисления $z = x * b$ следует умножить x на 3 и затем удвоить результат w , получая таким образом z . С точки зрения арифметики это совершенно правильно. С другой стороны, процедура P1 утверждает, что умножение x на 3 дает x , а процедура P2 утверждает, что в результате умножения любого числа на 2 всегда получается 24 . Эти два утверждения бессмысленны. Несмотря на это в данном контексте результаты вызовов процедур P1 и P2 дают ошибки, которые, как оказывается, компенсируют друг друга, так что общей ошибки не возникает.

Предположим, далее, что целевое утверждение изменилось на

$$G' : ? \text{умножить}(5, 2, z)$$

Как устанавливается спецификацией, G' имеет одно решение $z := 10$. Однако программа (с помощью процедуры P2) вычислит только $z := 24$, и, следовательно, она не является ни частично правильной, ни полной. Интуитивно неприемлемо, чтобы ранее правильная программа становилась неправильной лишь в результате постановки другого запроса при тех же самых утверждениях, описывающих проблемную область.

Рассмотрим теперь следующую программу, в целевом утверждении которой спрашивается, какие элементы списка L отрицательны.

Программа 31

$$\begin{aligned} G &: ? \exists(u, i, L), u < 0 \\ P1 &: \exists(10, 1, L) \\ P2 &: \exists(30, 3, L) \end{aligned}$$

Пусть спецификация S утверждает, что L есть список $(10, 20, 30, 40)$. Этого можно достичь, например, включив в S следующее утверждение:

$$\exists(u, i, L) \text{ тогда и только тогда, когда } (u = 10, i = 1) \vee (u = 20, i = 2) \vee (u = 30, i = 3) \vee (u = 40, i = 4)$$

Тогда в соответствии с S целевое утверждение G не имеет специфицируемых решений, поскольку все элементы списка L неотрицательны. Более того, данная программа не вычислит никаких решений и потому должна считаться полностью правильной. С другой стороны, мы можем видеть, что ее процедуры описы-

вают список L не полностью, поскольку в них пропущена информация о втором и четвертом элементах из L . Интуитивно неприемлемо, чтобы в результате исполнения программы получалось (верное) решение несмотря на то что не были проверены все элементы списка L .

Эти простые примеры показывают, что установленные ранее минимальные критерии правильности программ должны быть усилены за счет наложения дополнительных условий на их процедуры. На самом деле для устранения тех аномалий, которые только что обсуждались, достаточно всего двух условий. Мы опишем их как «достаточные условия», имея в виду тот факт, что они являются лишь достаточными, но не необходимыми условиями, гарантирующими абсолютную правильность.

Достаточное условие 1 частичной правильности программы (P, G) относительно спецификации S

Если $S \models P$, то (P, G) частично правильна относительно S

Другими словами, если процедуры логически следуют из спецификации, то программа будет частично правильной независимо от того, какое выбрано целевое утверждение.

Доказательство. Допустим, что выполняется условие $S \models P$, и пусть $t\Theta$ — произвольное вычисляемое решение целевого утверждения $?R^*(t)$. Тогда $P \models R^*(t\Theta)$. В силу транзитивности логического следования отсюда вытекает, что

для всех Θ , если $S \models P$ и $P \models R^*(t\Theta)$
то $S \models R^*(t\Theta)$

или, в эквивалентной форме,

если $S \models P$, то
(для всех Θ , если $P \models R^*(t\Theta)$, то $S \models R^*(t\Theta)$)

Это как раз и означает, что если имеет место $S \models P$, то выполняется также условие частичной правильности из определения 4.

Это более сильное требование $S \models P$ устранило бы тогда бесмысленные процедуры $P1$ и $P2$ из программы 30, поскольку ни та, ни другая не следуют логически из S . Разумеется, данная программа показывает также, что условие $S \models P$ не является необходимым для частичной правильности, т. е. само оно не следует из этого свойства.

Достаточное условие 2 полноты программы (P, G) относительно спецификации S

Если множество процедур P полно относительно S ,
то P является G -полным относительно S

Другими словами, если P является полным в смысле определения 6 (т. е. P полностью описывает содержимое специфицируемого отношения R^*), то множества P , конечно же, достаточно для вычисления всех решений, требуемых каждым конкретным целевым утверждением G .

Доказательство. Допустим, что множество процедур P является полным относительно S , и, следовательно,

для всех T , если $S \models R^*(T)$, то $P \models R^*(T)$

Отсюда вытекает, что для каждого вычисляемого решения $T = t\Theta$ целевого утверждения $G : ?R^*(t)$

если $S \models R^*(t\Theta)$, то $P \models R^*(t\Theta)$

Это заключение имеет место для любых подстановок Θ , и поэтому, добавляя к нему префикс «для всех Θ », мы получаем условие (определение 5) G -полноты P относительно S .

Сформулированное только что более сильное требование — P должно быть полным — устранило бы пропуски в программе 31, и процедуры теперь должны были бы давать описание элементов, находящихся на второй и четвертой позициях в списке L . Заключение о том, что целевое утверждение не имеет решений (что совершенно верно) было бы сделано тогда только после просмотра всех элементов списка L . Программа 31, заметим, показывает, что полнота P не является необходимым условием полноты (P, G) .

Достаточное условие 3 полной правильности программы (P, G) относительно спецификации S

Если $S \models P$ и множество процедур P является полным
то программа (P, G) полностью правильна относительно S

Доказательство. Из $S \models P$, в силу достаточного условия 1, следует частичная правильность, а из полноты P , в силу достаточного условия 2, следует полнота программы (P, G) ; стало быть, по определению 7, программа (P, G) является полностью правильной.

Заметим, что этот критерий не зависит от целевого утверждения; в нем просто требуется, чтобы множество процедур P не содержало ничего ложного о проблемной области и полностью описывало специфицируемое отношение. Если P удовлетворяет приведенному критерию, то тем самым устанавливается полная правильность целого класса программ (P, G) при любом возможном выборе целевого утверждения G вида $?R^*(t)$.

Требования, сформулированные в этом достаточном условии и гарантирующие полную правильность, интуитивно осмыслены и имеют важное методологическое значение. Никогда не может быть хороших причин для написания программ, подобных программе 30, в которой используются заведомо неверные утверждения. Иногда, однако, может быть оправдано составление неполного множества процедур Р. Можно было бы, например, довольствоваться вычислением только некоторых решений и для уменьшения стоимости исполнения программы опустить процедуры, способные найти другие решения. Этот путь более экономичен, чем возможность написания полного множества процедур и вставления в него затем операторов отсечения для удаления нежелательных решений, поскольку он ведет к выигрышу как в памяти, так и во времени исполнения программы. С другой стороны, использование неполного множества процедур может усложнить интерпретацию исполнения программы, не дающего решений, а также исполнения, основанного на принципе отрицание как неудача. В общем случае гораздо лучше пользоваться полными множествами процедур, если только какие-либо причины не вынуждают поступать иначе.

Подобно тому как частичную правильность можно устанавливать, показывая, что $S \models P$, так и полноту Р при некоторых обстоятельствах можно продемонстрировать, доказав, что $P \models S$, поскольку отсюда вытекает условие полноты из определения 6. На практике, однако, это редко бывает возможно: множества логических процедур даже при условии их полноты содержат, как правило, меньший объем информации, чем спецификации, которым они соответствуют. Мы можем проиллюстрировать сказанное на следующем простом примере. Пусть R^* — отношение $\{(A, B), (C, B)\}$, специфицируемое в S предложением

S1 : $R^*(x, y)$ тогда и только тогда, когда $(x = A, y = B) \vee (x = C, y = B)$

Допустим также, что в оставшейся части S дается лишь некоторое полное определение отношения тождества ($=$). Тогда следующее множество процедур

$R^*(x, y)$ если $x = A, y = B$

$R^*(x, y)$ если $x = C, y = B$

$A = A$

$B = B$

$C = C$

является полным, поскольку каждая пара (x, y) из R^* может быть вычислена с помощью Р. Тем не менее из Р не следуют предложения, содержащиеся в S. Эта «слабость» Р объясняется

тем фактом, что хотя множество процедур P логически влечет часть «только тогда» предложения $S1$

если $(x = A, y = B) \vee (x = C, y = B)$, то $R^*(x, y)$

из него совсем не следует часть «тогда»

если $R^*(x, y)$, то $(x = A, y = B) \vee (x = C, y = B)$

и, стало быть, из P не может следовать $S1$. Точно так же из P нельзя вывести многие отношения тождества (типа $D = D$) определяемые спецификацией S . Как правило, в процедурах из множества P будет отсутствовать много информации из S , поскольку в вычислительных целях она не используется. Исходя из множества процедур P , можно вычислить все отношение R^* и при этом процедурам из P не требуется знать, что это имеет место. Полнота P должна быть установлена другими средствами. Мы опишем их в разд. V. 7.

V. 4. Разрешимость

Программа (P, G) является *разрешимой*, когда с ее помощью вычисляется по крайней мере одно решение, т. е. когда ее пространство вычислений содержит хотя бы одно успешное вычисление. Это свойство можно определить формально следующим образом.

Определение 8. Разрешимость программы $(P, G : ? R^*(t))$.

Программа (P, G) разрешима тогда и только тогда, когда $P \models R^*(t\Theta)$ для некоторой подстановки Θ .

Разрешимость — это свойство исключительно программы, никак не связанное с какой-либо спецификацией. Хотя разрешимость обычно не включается в условия полной правильности, иногда может оказаться полезным в ходе процесса верификации попутно исследовать и это свойство.

Например, для установления разрешимости программы 29 нужно было бы показать, что

для некоторой подстановки Θ , $P \models$
подмнож $(w : A : \emptyset, A : B : C : \emptyset)\Theta$

т. е.

$P \models (\exists w) \text{подмнож}(w : A : \emptyset, A : B : C : \emptyset)$

Разумеется, один из способов достижения этой цели состоял бы просто в исполнении программы 29! Однако в силу объясняемых ниже (разд. V. 5) причин в результате исполнения разрешимой программы может и не быть получено никаких решений, и поэтому данный подход, вообще говоря, ненадежен. Во всяком

случае в ходе исполнения программы всегда делается попытка определить действительные значения целевых переменных, а не просто доказывается, что такие значения существуют. И, как правило, для решения первой задачи требуются значительно большие усилия. В особенности сказанное относится к программам, содержащим рекурсивные или итеративные процедуры. В соответствии с обсуждаемым подходом для доказательства разрешимости такой программы должны быть на самом деле выполнены все шаги, задаваемые этими процедурами. Были, однако, разработаны другие методы демонстрации разрешимости, которые позволяют сделать заключение о результатах итеративных и рекурсивных вычислений, в действительности их не порождая. В этих методах обычно требуется расширить знания, закодированные в процедурах из P , первоначально аксиомами индукции, и тогда этой информации оказывается достаточно для доказательства того, например, что итерация через конечное число шагов обязательно сойдется к некоторому решению. Такого рода доказательства называются «неконструктивными доказательствами существования», поскольку в отличие от резолютивных доказательств находить решения, существование которых устанавливается, в них не нужно¹⁾.

Иногда разрешимость программы можно установить более просто путем анализа ее спецификации S . Пусть уже известно, что программа (P, G) является полной. Тогда (P, G) будет разрешимой, если

$$\text{для некоторой подстановки } \Theta, S \models R^*(t\Theta)$$

(обратное, заметим, неверно), что с очевидностью вытекает из определений 5 и 8. Практическое значение этого утверждения состоит в том, что целевое утверждение $R^*(t\Theta)$, может быть, гораздо легче доказать, исходя из спецификации, непосредственно описывающей проблемную область, чем из множества ориентированных на вычисления процедур. Данный метод имеет еще одно преимущество: он подтверждает, что вычисляемое решение $t\Theta$ не только существует, но и является также правильным относительно S .

Точно так же анализ спецификации S может быть наиболее простым способом доказательства *неразрешимости*. Пусть известно уже, что программа (P, G) частично правильна. Тогда

¹⁾ Имеются в виду, конечно, резолютивные доказательства из множеств хорновских дизъюнктов. Если же мы будем доказывать существование исходя из произвольной спецификации S , то резолютивный вывод может оказаться неконструктивным и решения не дать. — Прим. перев.

(P, G) неразрешима, если

для всех подстановок Θ $S \models \neg R^*(t\Theta)$

(обратное, заметим, неверно). Установить это, может быть, проще, чем, используя процедуры из P и строгое определение неразрешимости, показывать, что

для всех подстановок Θ не $(P \models R^*(t\Theta))$

Данные применения спецификации S, как и все другие, основываются на допущении непротиворечивости S.

V.5. Правильность алгоритмов: определения

В этом разделе мы изучим свойства правильности логических алгоритмов. Логический алгоритм получается в результате выбора некоторой управляющей стратегии С для управления исполнением некоторой логической программы (P, G); логический алгоритм можно обозначить поэтому тройкой (P, G, С). Так как тройка (P, G, С) содержит также полную информацию, необходимую для определения всех деталей работы алгоритма, то можно считать, что она в то же время является обозначением исполнения программы. Таким образом, мы будем использовать здесь термины алгоритм и исполнение как равнозначные.

В каждой из существующих реализаций основной составной частью стратегии С является, как правило, стандартная стратегия управления, которая регулирует выбор вызовов и отвечающих на них процедур и в пространстве вычислений программы осуществляет поиск в глубину. В предыдущих главах мы уже видели, что эта базовая стратегия может быть разными способами усиlena или ограничена с помощью других управляющих механизмов, таких как порождение лемм, отрицание как неудача, операторы отсечения и т. д. Для того чтобы упростить представление наиболее важных моментов в алгоритме верификации, в дальнейшем указанные модификации рассматриваться не будут, и наш анализ поэтому будет в основном ограничен алгоритмами, управляемыми одной лишь стандартной стратегией.

V.5.1. Производимость

Целью исполнения программы является порождение вычислений. В дальнейшем для обозначения вычислений мы будем использовать букву Г (гамма). Если предположить, что С — стандартная стратегия для исполнения методом сверху вниз, то каждое вычисление Г будет представлять собой последователь-

ность целевых утверждений, начинающуюся с исходного целевого утверждения программы G .

При конкретном управлении C всякое вычисление можно классифицировать либо как *производимое* (producible), либо как *непроизводимое*. Эти понятия определяются следующим образом.

Определение 9. Производимость вычисления Γ из программы (P, G) при управлении C .

Вычисление Γ является производимым из программы (P, G) при управлении C тогда и только тогда, когда выполнение (P, G, C) порождает каждое целевое утверждение из Γ за конечное число шагов (т. е. в течение некоторого конечного времени с момента начала исполнения программы); в противном случае вычисление Γ называется непроизводимым при управлении C .

Это определение применяется не только к стандартной стратегии (поиска в глубину), но и к любой другой стратегии управления методом сверху вниз.

В соответствии со следующим определением понятие производимости можно с успехом применять к вычисляемым решениям так же, как к вычислениям.

Определение 10. Производимость решения T из программы (P, G) при управлении C .

Вычисляемое решение T является *производимым* из программы (P, G) при управлении C тогда и только тогда, когда оно вычисляется некоторым производимым при C вычислением (т. е. решение T вычислимо за некоторое конечное время); в противном случае решение T называется *непроизводимым*.

Цель этих определений — прояснить разницу между тем, что логически вычислимо в соответствии с программой, и тем, что действительно вычислимо (производимо), когда программа исполняется конкретным способом. В дальнейшем будет показано, что не все вычислимые решения обязательно будут производимыми даже при стандартной стратегии; будут они таковыми или нет, зависит частично от управления, а частично — от структуры полного пространства вычислений программы. Все это станет более понятным из следующего раздела, где рассматривается несколько конкретных примеров.

V.5.2. Сравнение трех алгоритмов

Приводимая ниже программа представляет собой довольно искусственный пример, которого будет достаточно для иллюстрации всех интересующих нас сейчас свойств алгоритмов. Мы

не будем пытаться придать этой программе какое-либо практическое значение.

Программа 32

$$\begin{array}{ll} G : & ? \ g(y) \\ P1 : & g(y) \text{ если } p(y), q(y) \\ P2 : & p(A) \\ P3 : & q(A) \\ P4 : & q(f(x)) \text{ если } q(f(f(x))) \end{array}$$

Процедуры P1—P4 образуют полное множество процедур P. Допустим теперь, что эта программа исполняется с помощью стандартной стратегии и при указанном текстуальном упорядочении процедур и вызовов — два этих аспекта управления составляют общее управление C1. Получаемый в результате алгоритм (P, G, C1) порождает следующее единственное вычисление Γ_1 .

$$\begin{array}{l} \Gamma_1 : ? \ g(y) \\ ? \ p(y), q(y) \\ ? \ q(A) \\ ? \quad \text{решение } y := A \end{array}$$

Это вычисление является конечным, производимым, успешным; оно дает решение $y := A$. Найденное решение является вычисляемым решением, поскольку $P \models g(A)$, а также производимым решением, поскольку оно действительно получается через конечное число шагов исполнения программы. Данный алгоритм совершенно безобиден и сравним с многими уже рассмотренными до сих пор примерами.

Теперь посмотрим, что получится в результате изменения текстуального порядка двух вызовов из тела процедуры P1 на обратный. Логическое содержание программы при этом не изменится и поэтому программа (P, G) останется той же самой. Однако эти действия приведут к изменению управления: оно будет эквивалентно управлению, которое получится, если процедуру P1 оставить без изменений и взять правило вычислений, выбирающее последний вызов из каждого целевого утверждения вместо первого. С какой бы точки зрения это ни рассматривать, новое общее управление C2 отличается от C1. Получающийся в результате алгоритм (P, G, C2) порождает следующие два вычисления:

$$\begin{array}{l} \Gamma_{2a} : ? \ g(y) \\ ? \ q(y), p(y) \\ ? \ p(A) \\ ? \quad \text{решение } y := A \end{array}$$

$$\begin{array}{l} \Gamma_{2b} : ? \ g(y) \\ ? \ q(y), p(y) \\ ? \ q(f(f(x))), p(f(x)) \\ ? \ q(f(f(f(x)))), p(f(x)) \\ \text{и т. д. до бесконечности} \end{array}$$

Вычисление Γ_{2a} является конечным, производимым, успешным и дает решение $y := A$. Оно порождается первым. Другое вычисление Γ_{2b} оказывается бесконечным и, стало быть, хотя и является производимым, но не дает соответствующего производимого решения.

Рассмотрим, иаконец, третий алгоритм, который получается в результате двух текстуальных изменений в программе 32: первое меняет порядок двух вызовов из P1 на обратный (как и во втором алгоритме), а второе переставляет местами процедуры P3 и P4. Таким образом, у нас опять имеется та же самая программа (P, G), но мы используем новое общее управление C3. Алгоритм (P, G, C3) начнет строить вычисление Γ_{2b} . Поскольку оно бесконечно, а стандартная стратегия осуществляет поиск в глубину, выполнение программы будет все время занято построением вычисления Γ_{2b} , и, следовательно, вычисление Γ_{2a} становится непроизводимым.

Эти отличающиеся друг от друга поведения можно проще всего понять, рассматривая полное пространство вычислений, определяемое программой (P, G), в предположении, что выполнение ведется методом сверху вниз. Пространство состоит как раз из трех вычислений Γ_1 , Γ_{2a} и Γ_{2b} . Мы видели в гл. II, что различные правила вычислений для выбора вызова соответствуют различным подпространствам полного пространства вычислений. Это ясно показывают три приведенных выше алгоритма. В первом посредством включения в управление C1 программой 32 стандартного правила вычисления слева направо выделяется подпространство $\{\Gamma_1\}$. В двух оставшихся алгоритмах эффективное включение в управления C2 и C3 программой 32 правила вычисления справа налево приводит к выделению одного и того же подпространства $\{\Gamma_{2a}, \Gamma_{2b}\}$. Каждое из выбранных подпространств покрывает полное множество решений $\{y := A\}$.

Изменение текстуального порядка процедур в программе эквивалентно принятию другого правила поиска для выбора процедур. Снова, как было показано в гл. II, каждому такому правилу поиска соответствует своя последовательность порождения вычислений из выбранного подпространства. Правило поиска, использованное в управлении C1, может порождать только одно вычисление Γ_1 , тогда как в C2 это же самое правило может порождать два вычисления и порождает их в указанном выше порядке. Однако изменение порядка следования процедур P3 и P4 эквивалентно изменению стандартного правила поиска (с первой процедуры до последней) на обратное, и поэтому управление C3 исследует то же подпространство, что и C2, но в обратном порядке.

В заключение мы отметим следующие основные положения: (i) наличие или отсутствие бесконечных вычислений зависит только от логического содержания программы (P, G); (ii) содержится или нет имеющиеся бесконечные вычисления в выбранном подпространстве, определяется общим управлением; (iii) общее управление определяет, будут или нет бесконечные вычисления из выбранного подпространства порождаться перед некоторыми или всеми конечными вычислениями; (iv) логически вычислимое решение не обязано являться производимым при каждом конкретном управлении — оно будет производимым только тогда, когда в течение конечного времени в ходе исполнения программы будет выбрано и полностью построено вычисление, соответствующее этому решению.

Ввиду (iv) теперь должно быть понятно, почему, как утверждалось в разд. V.4, для проверки разрешимости программы не всегда бывает достаточно ее выполнить — если сначала будет исследоваться какая-либо бесконечная область пространства вычислений, то никаких решений в результате исполнения найти не удастся.

Возможность существования логически вычислимых решений, не являющихся производимыми, появляется всякий раз, когда используемая стратегия управления «*несправедлива*», т. е. когда ее внимание не распределяется поровну между всеми имеющимися вычислениями. Последовательная стратегия поиска в глубину неизбежно несправедлива в силу того, что она обязана построить до конца текущее вычисление прежде, чем перейти к следующему.

Наоборот, последовательная стратегия поиска в ширину, согласно которой по очереди выполняется по одному шагу в каждом из вычислений, и, таким образом они порождаются почти параллельно, — эта стратегия по существу «*справедлива*», и потому все конечные вычисления (а следовательно, и все вычисляемые решения) обычно порождаются при ней за конечное время. Исключение может возникнуть лишь тогда, когда в процессе поиска мы сталкиваемся с бесконечным числом вычислений, поскольку в этом случае частично построенное успешное вычисление может навсегда остаться незавершенным, в то время как интерпретатор будет занят применением текущего расширяющего шага ко всей оставшейся бесконечной совокупности вычислений. Как правило, такого рода пространства вычислений появляются в тех случаях, когда в программах вызываются встроенные процедуры, которые ведут себя так, как будто они реализованы посредством бесконечного множества фактов.

Вычисляемые решения могут стать непроизводимыми, даже если программа (P, G) определяет конечное пространство вы-

числений (содержащее конечное число конечных вычислений). Такое может произойти, когда управление С способно удалять вычисления в ходе исполнения программы, реагируя, например, на оператор отсечения или на директивы вычеркивания процедур. Фактически это эквивалентно управлению исполнением посредством стратегии *неполного* поиска, т. е. такой стратегии, при которой все логически определяемые программой (P, G) возможности полностью не исследуются.

V.5.3. Завершаемость

Завершаемость является простым понятием, когда мы применяем его к программе, написанной на традиционном детерминистском языке: такая программа предлагает только одно возможное вычисление (после того как входные данные, если они имеются, уже определены), и ее исполнение завершается в том и только том случае, когда это вычисление конечное.

Понятие завершаемости становится потенциально более сложным, когда мы применяем его к логическим алгоритмам, предлагающим несколько вычислений. В этом случае в зависимости от выбранного определения для завершения алгоритма (P, G, C) может потребоваться, например, (i) чтобы все успешные вычисления были производимыми или (ii) чтобы вычисления всех видов были производимыми и чтобы их было лишь конечное число. Условие (i) является более слабым по сравнению с условием (ii), так как в нем требуется только то, чтобы все вычисляемые решения были получены за конечное время, даже если потом исполнение программы будет продолжаться бесконечно долго, либо попав в ловушку бесконечного вычисления, либо оказавшись вынужденным исследовать бесконечную совокупность неудачных вычислений. Условие (ii) является более сильным, поскольку в нем утверждается, что весь процесс исполнения программы должен завершиться за конечное время.

На интуитивном уровне условие (ii) кажется более подходящим, если, конечно, мы согласимся с тем, что ему не будут удовлетворять некоторые виды программ, оказывающихся тем не менее практически полезными. Это такие программы, которые ведут себя так, как будто они собираются работать бесконечно долго. К ним относятся операционные системы, программы контроля процессов реального времени и т. д. Может возникнуть, например, желание выполнить следующую программу, которая контролирует состояние x некоторого процесса в моменты времени $t = 0, 1, 2, \dots$ и т. д.

Программа 33

? монитор(t)
 монитор(t) если состояние(x, t),
 дисплей(x, t), монитор($t + 1$)

и соответствующие процедуры для предиката
дисплей, причем вызовы **состояние** решаются
 непосредственно через интерфейс с аппаратными
 средствами этого процесса

Здесь моменты времени определяются программой монитор, и мы предполагаем, что каждый вызов **состояние** по каждому значению t выдает единственное значение для переменной x . В этом случае программа будет детерминированной, а ее единственное вычисление является бесконечным. Следовательно, исполнение программы никогда не даст решения исходного целевого утверждения и никогда не завершится. Напротив, желаемый результат — это последовательность значений (x, t) , которые сами являются единственными вычисляемыми решениями вызовов **состояние**.

Приблизительно такое же поведение можно получить, используя несколько иной стиль программирования, такой как в приводимой ниже программе.

Программа 34

? состояние(x, t), дисплей(x, t)
 и процедуры для предиката **дисплей**, причем вызовы
состояние решаются, как и прежде

Мы предполагаем здесь, что сам процесс, а не монитор, сообщает моменты времени t , а также состояния x . Эта программа недетерминированная, поскольку процесс ведет себя так, как если бы он был реализован с помощью бесконечного числа фактов **состояние**, по одному для каждого момента времени t . Успешные пары (x, t) выдаются теперь путем многократного выполнения процесса возврата, в результате чего по очереди вызывается каждый из (неявных) фактов **состояние**. Подпространство вычислений будет состоять теперь из бесконечного числа успешных вычислений и бесконечных вычислений содержать не будет. Исполнение программы снова не завершается.

Несмотря на наличие такого рода исключений, мы примем до некоторой степени произвольно указанное выше условие завершаемости (ii), формализуя его следующим образом.

Определение 11. Завершаемость исполнения (P, G, C)

Исполнение (P, G, C) завершается тогда и только тогда, когда оно даст конечное множество вычислений, причем все они имеют конечную длину.

Из этого определения следует, что алгоритмы, полученные в результате исполнения программ 33 и 34 при стандартной стратегии выполнения, являются незавершаемыми. Что касается программы 32, обсуждавшейся в предыдущем разделе, то, согласно данному определению, исполнение $(P, G, C1)$ завершается, тогда как оба исполнения $(P, G, C2)$ и $(P, G, C3)$ не завершаются.

V.5.4. Определения правильности логических алгоритмов

При традиционном подходе к доказательству правильности программ было принято включать условие завершаемости в определение полной правильности. Имея дело с логическими алгоритмами, мы не станем в этом отношении следовать указанному подходу, а сконцентрируем свое внимание лишь на правильности и полноте множества порождаемых решений. Завершаемость алгоритмов, а также разрешимость программ рассматриваются как важные, заслуживающие изучения свойства, но их не следует включать в необходимые условия правильности.

На содержательном уровне логический алгоритм (P, G, C) считается *полностью правильным* относительно спецификации в том и только том случае, когда он является и *частично правильным*, и *полным*. Логический алгоритм будет частично правильным тогда и только тогда, когда все производимые им решения являются специфицируемыми решениями, и он будет полным тогда и только тогда, когда все специфицируемые решения являются производимыми решениями. Другими словами, полностью правильный алгоритм выдает за конечное время в точности те решения, которые спецификация S приписывает целевому утверждению G .

Для того чтобы сформулировать эти условия в логических терминах, удобно использовать выражения вида

$$A \vdash_C B$$

означающие, что формула B выводима из формулы A с помощью резолюции, управляемой стратегией C . В частности, выражение

$$P \vdash_C R^*(t\Theta)$$

означает, что решение $t\Theta$ предполагаемого целевого утверждения $G : ?R^*(t)$ является производимым при стратегии C . Необходимые условия правильности логических алгоритмов можно сформулировать теперь следующим образом.

Определение 12. Частичная правильность (P, G, C) относительно спецификации S .

Логический алгоритм (P, G, C) является частично правильным относительно спецификации S тогда и только тогда, когда для всех Θ , если $P \vdash_c R^*(t\Theta)$, то $S \models R^*(t\Theta)$

Определение 13. Полнота (P, G, C) относительно спецификации S .

Логический алгоритм (P, G, C) является полным относительно спецификации S тогда и только тогда, когда

для всех Θ , если $S \models R^*(t\Theta)$, то $P \vdash_c R^*(t\Theta)$

Определение 14. Полная правильность (P, G, C) относительно спецификации S .

Логический алгоритм (P, G, C) является полностью правильным относительно спецификации S в том и только том случае, когда

для всех Θ $S \models R^*(t\Theta)$ тогда и только тогда, когда
 $P \vdash_c R^*(t\Theta)$

Приведенные определения по своей структуре подобны (и это весьма приятно) дававшимся ранее определениям правильности программ (определения 4, 5 и 7), что является следствием использования двойственных понятий (логической) вычислимости и (действительной) производимости, позволяющих отличать результаты программ от результатов алгоритмов.

V. 6. Правильность алгоритмов: достаточные условия

Верификацию логических алгоритмов можно упростить с помощью применения достаточных условий. Как и при рассмотрении верификации программ, эти условия не только оказываются логически более сильными, чем необходимые, но позволяют также выявить разумные с методологической точки зрения требования, которые следовало бы предъявлять к логическим алгоритмам. Достаточные условия можно сформулировать следующим образом.

Достаточное условие 4 частичной правильности алгоритма (P, G, C) относительно спецификации S .

Если программа (P, G) частично правильна, то логический алгоритм (P, G, C) также является частично правильным.

Это условие должно быть достаточно очевидным: если программа (P, G) частично правильна, то каждое вычисляемое решение $t\Theta$ будет специфицируемым решением, а при условии, что стратегия C основана на корректной системе логического выво-

да, такой как резолюция, каждое производимое решение $t\Theta$ является вычисляемым решением; отсюда сразу вытекает частичная правильность алгоритма (P, G, C) .

Достаточное условие 5 полноты алгоритма (P, G, C) относительно спецификации S .

Если программа (P, G) полна и определяет конечное пространство вычислений, а стратегия C исчерпывающая, то логический алгоритм (P, G, C) является полным.

Это условие также понятно: если пространство вычислений конечно, а управление C исследует все вычисления в каждом выбранном подпространстве, то все вычисляемые решения являются производимыми; поэтому, если программа (P, G) полна, то все специфицируемые решения будут вычисляемыми и, следовательно, производимыми, что и доказывает полноту алгоритма (P, G, C) . Управление C будет исчерпывающим, если оно совпадает со стандартной стратегией, в которой не используются операторы отсечения и другие подобные директивы.

Достаточное условие 6 полной правильности алгоритма (P, G, C) относительно спецификации S .

Если программа (P, G) полностью правильна и определяет конечное пространство вычислений, а управление C исчерпывающее, то логический алгоритм (P, G, C) является полностью правильным.

Этот критерий вытекает непосредственно из двух предыдущих достаточных условий и определения 14. С помощью приведенного ранее достаточного условия 3 полной правильности программы (P, G) можно сформулировать даже еще более сильное условие полной правильности алгоритма (P, G, C) .

Достаточное условие 7 полной правильности алгоритма (P, G, C) относительно спецификации S .

Если S логически влечет P , множество процедур P является полным, программа (P, G) определяет конечное пространство вычислений, а управление C исчерпывающее, то логический алгоритм (P, G, C) является полностью правильным.

Мы воспользуемся этим критерием в следующем разделе, где приводится общая схема верификации программы 29 (программы подмнож) и ее исполнения при стандартной стратегии управления.

V.7. Пример верификации

Применение теории верификации мы продемонстрируем теперь в общих чертах на примере программы 29. Тем самым мы покажем, что обсуждавшиеся выше идеи, носящие скорее теоретиче-

ский характер, могут быть достаточно прямым способом использованы для верификации программ. Мы заново сформулируем все утверждения программы 29 и для удобства ссылок снабдим их метками.

Программа 29

```

G : ? подмнож( $w : A : \emptyset, A : B : C : \emptyset$ )
P1 : подмнож( $x, y$ ) если пустое( $x$ )
P2 : подмнож( $v : x', y$ ) если  $v \in y$ , подмнож( $x', y$ )
P3 :  $v \in v : z$ 
P4 :  $v \in u : z$  если  $v \in z$ 
      а также встроенные процедуры для
      предиката пустое
  
```

Мы покажем, что и сама эта программа и любое ее исполнение при стандартной стратегии управления являются полностью правильными относительно заданной спецификации.

V.7.1. Выбор критериев

Для того чтобы проверить правильность программы, нужно решить, какими критериями воспользоваться — необходимыми или достаточными. Вообще говоря, есть два соображения в пользу выбора последних. Первое из них заключается в том, что с помощью достаточных критериев проверяются три свойства программ, а именно:

- (i) Р логически следует из спецификации S;
- (ii) Р является полным относительно S;
- (iii) программа (Р, G) определяет конечное пространство вычислений;

Можно было бы ожидать, что эти свойства выполняются для составленных в хорошем стиле логических программ, и, конечно, разумно попытаться сначала доказать те свойства, выполнение которых кажется наиболее вероятным. Во-вторых, достаточные условия легче формулировать и исследовать, чем необходимые. В дальнейшем для подтверждения полной правильности программы (Р, G) устанавливаются первые два из приведенных выше свойств. Затем будет установлено третье свойство и, таким образом, гарантируется, что стандартное исполнение программы 29 даст за конечное время в точности те правильные примеры w , которые делают множество $\{w\}$ подмножеством $\{A, B, C\}$.

V.7.2. Выбор спецификации

Выбранная спецификация S состоит из множества предложений, записанных в общем языке логики первого порядка. Ниже мы приводим наиболее важные из них. В первом предложении формулируется стандартное определение предиката подмнож; во втором определяется отношение принадлежности для принятого представления множеств (в виде структурированных термов), а в третьем дается определение пустого множества. Кроме того, в эту спецификацию (неявно) входит полное описание отношения тождества ($=$).

Спецификация для программы 29

- $$\begin{aligned} S1 : \text{подмнож}(x, y) &\Leftrightarrow (\forall u)(u \in y \text{ если } u \in x) \\ S2 : u \in x &\Leftrightarrow (\exists v \exists x')(x = v : x', (u = v \vee u \in x') \\ S3 : \text{пустое}(x) &\Leftrightarrow \neg (\exists v \exists x')(x = v : x') \\ &\text{и полное описание отношения } = \end{aligned}$$

В этих предложениях символ \Leftrightarrow является сокращением для выражения «тогда и только тогда, когда».

V.7.3. Преобразование определяющих

Каждое предложение в приведенной выше спецификации намеренно представлено в виде

определяемое \Leftrightarrow определяющее

где в качестве *определяемого* (т. е. объекта, который определяется) выступает единственный предикат $g()$, именующий некоторое специфицируемое отношение g , а в качестве *определяющего* (т. е. определяющего свойства объекта) — произвольная формула. Этот стандартный стиль представления спецификации объясняется следующим образом. Пусть можно показать, что из спецификации S логически следует некоторое предложение вида $D \Leftrightarrow D'$, где D есть определяющее для специфицируемого отношения g . Тогда отсюда вытекает, что D' — это другое, эквивалентное первому определяющее для g . Более формально, мы утверждаем, что

если $S \models g() \Leftrightarrow D$ и $S \models D \Leftrightarrow D'$
то $S \models g() \Leftrightarrow D'$

Замена предложения D на D' в определении отношения g называется *преобразованием определяющего*. Применяя последовательно такие преобразования, можно получить *доказательство*

ство, в котором на каждом шаге образуется новое определение отношения Γ , логически следующее из спецификации S . Доказательство будет иметь такой вид:

- $\Gamma(\) \Leftrightarrow D_1$ (определение, данное изначально в S)
- $\Gamma(\) \Leftrightarrow D_2$ (используя $S \models D_1 \Leftrightarrow D_2$)
- $\Gamma(\) \Leftrightarrow D_3$ (используя $S \models D_2 \Leftrightarrow D_3$)
- и т. д.

Цель построения такого доказательства состоит в получении какого-либо n -го определения

$$\Gamma(\) \Leftrightarrow D_n$$

в котором предложение D_n является некоторой дизъюнкцией $B_1 \vee B_2 \vee \dots \vee B_m$, а каждый ее член B_i имеет синтаксис тела процедуры. Если это так, то отсюда непосредственно вытекает, что S логически влечет следующее множество из m выводимых процедур:

$$\begin{aligned} \Gamma(\) &\text{ если } B_1 \\ \Gamma(\) &\text{ если } B_2 \\ &\vdots \\ \Gamma(\) &\text{ если } B_m \end{aligned}$$

В ходе выполнения верификации, основанной на достаточных критериях, доказательства такого рода строятся для каждого отношения, встречающегося в программе и отличного от тех, которые реализованы встроенными процедурами. Поэтому в рассматриваемом примере будут нужны два доказательства: одно для отношения **подмнож** и другое для отношения принадлежности **€**. Доказательства «направляются» таким образом, чтобы выводились в точности те процедуры, которые входят в нашу программу. Если это будет успешно выполнено, то тем самым будет доказано, что программа (P, G) является полностью правильной. Ее частичная правильность устанавливается непосредственно тем фактом, что каждая процедура логически следует из S , а ее полнота вытекает из того факта, что каждое преобразование определяющих сохраняет эквивалентность, т. е. гарантируется, что каждый кортеж термов, описываемый предыдущим определяющим, будет описываться и последующим.

V.7.4. Вывод процедур

Ниже в общих чертах описывается применение преобразований определяющих для вывода процедур $P1$ и $P2$ для предиката **подмнож**. Ради краткости на каждом шаге показывается только

определяющее D_1 , а не все предложение подмнож(x, y) $\Leftrightarrow D_1$ целиком.

Доказательство процедур подмнож

- $$\begin{aligned}D_1 &: (\forall u)(u \in y \text{ если } u \in x) \\D_2 &: (\forall u)(u \in y \text{ если } (\exists v \exists x')(x = v : x', (u = v \vee u \in x'))) \\D_3 &: \neg(\exists v \exists x')(x = v : x') \vee \\&\quad (\exists v \exists x')(x = v : x', (\forall u)(u \in y \text{ если } u = v \vee u \in x')) \\D_4 &: \text{пустое}(x) \vee (\exists v \exists x')(x = v : x', (\forall u)(u \in y \text{ если } u = v), \\&\quad (\forall u)(u \in y \text{ если } u \in x')) \\D_5 &: \text{пустое}(x) \vee (\exists v \exists x')(x = v : x', v \in y \text{ подмнож}(x', y))\end{aligned}$$

На этом этапе каждый член дизъюнкции из определяющего (D_5) становится конъюнкцией предикатов и является, стало быть, правильно построенным телом процедуры. Заметим, что наличие кванторов существования в префикссе какого-либо из этих дизъюнктивных членов не играет никакой роли, поскольку каждое выводимое предложение

A если ($\exists z$) B (z), где переменная z не входит в **A**
эквивалентно предложению

$$(\forall z)(A \text{ если } B(z))$$

которое имеет вид стандартной процедуры. Таким образом, из определяющего D_5 мы можем теперь вывести два следующих предложения, соответствующих двум его дизъюнктивным членам.

- $$\begin{aligned}P1 &: \text{подмнож}(x, y) \text{ если пустое}(x) \\&\quad \text{подмнож}(x, y) \text{ если } x = v : x', v \in y, \text{ подмнож}(x', y)\end{aligned}$$

С помощью элементарных свойств предиката $=$ второе предложение можно упростить и получить тем самым процедуру

$$P2 : \text{подмнож}(v : x', y) \text{ если } v \in y, \text{подмнож}(x', y)$$

Итак, выведены обе процедуры подмнож из программы 29. Шаги доказательства можно неформально объяснить следующим образом.

Шаг 1. D_1 преобразуется в D_2 путем замены подформулы $u \in x$ на ее определяющее в соответствии с S2.

Шаг 2. D_2 преобразуется в D_3 с помощью обобщения такого правила:

формула **A если($\exists z$) (B(z), C(z))** логически эквивалентна

формуле $\neg(\exists z)B(z) \vee (\exists z)(B(z), (A \text{ если } C(z)))$ ¹⁾

Шаг 3. D_3 преобразуется в D_4 с помощью правила

формула $A \text{ если } B \vee C$ логически эквивалентна
формуле $(A \text{ если } B), (A \text{ если } C)$

для того, чтобы расщепить подформулу $(\forall u)$ (и т. д.) формулы D_3 на две конъюнкции.

Шаг 4. Наконец, первая из полученных конъюнкций упрощается до эквивалентной формулы $v \in y$, а вторая в соответствии с $S1$ заменяется на определяемый ею предикат подмнож(x', y), в результате чего получается D_5 .

Последовательность (D_1, \dots, D_5) представляет собой только основные этапы доказательства. Вообще говоря, каждый шаг содержит некоторое произвольное количество внутренних манипуляций либо с одним лишь текущим определяющим (как, например, при выводе D_4 из D_3), либо привлечением еще некоторого предложения из спецификации S (как, например, при выводе D_5 из D_4). Во всех этих манипуляциях используются разнообразные правила вывода в логике предикатов первого порядка. Для успешного и эффективного выполнения верификации вручную требуется определенный опыт в обращении с подобными правилами. Мы не пытаемся привести здесь исчерпывающую коллекцию полезных для верификации программ правил вывода. Вместо этого (в разд. V.9) интересующемуся читателю рекомендуется обратиться к стандартным книгам по математической логике и исследовательским статьям, в которых описываются правила, оказавшиеся наиболее полезными на практике.

Следующая задача в рассматриваемом примере заключается в верификации процедур $P3$ и $P4$ для предиката \in . Доказательство начинается с определяющего отношения $u \in x$, содержащегося в предложении $S2$, а именно

$$D_1 : (\exists v \exists x')(x = v : x', (u = v \vee u \in x'))$$

¹⁾ Здесь автор допустил неточность. Эквивалентными эти формулы не являются, ибо, как нетрудно видеть, первая формула не следует логически из формулы

$$(\exists z)(B(z), (A \text{ если } C(z)))$$

Тем не менее, требуемую для преобразования D_2 в D_3 эквивалентность (т. е. приведенное выше правило, в котором вместо A , B и C подставлены соответствующие подформулы из D_2) можно вывести из спецификации S с учетом присутствующего в ней определения отношения $=$. — *Прим. перев.*

На первом шаге правило

формула $A, (B \vee C)$ логически эквивалентна формуле
 $(A, B) \vee (A, C)$

используется для того, чтобы преобразовать D_1 в предложение
 $D_2 : (\exists v \exists x')((x = v : x', u = v) \vee (x = v : x', u \in x'))$

из которого мы можем вывести две процедуры

$u \in x$ если $x = v : x', u = v$

$u \in x$ если $x = v : x', u \in x$

В соответствии с предполагаемыми свойствами отношения тождества ($=$) их можно упростить до процедур

$v \in v : x'$
 $u \in v : x' \text{ если } u \in x'$

которые с точностью до переименования переменной z на x' совпадают с процедурами Р3 и Р4 из программы 29, что и требовалось.

На этом верификация программы (P, G) завершается: доказано, что она полностью правильна относительно данной спецификации S . Более того, она останется полностью правильной при любом другом целевом утверждении, поскольку в наших рассуждениях, основанных на достаточных критериях, целевое утверждение G не учитывалось.

V.7.5. Доказательство завершаемости

Завершаемость исполнения (P, G, C) программы 29 устанавливается здесь с помощью неформальных рассуждений. Рассмотрим произвольное вычисление Γ , которое логически определяется этой программой. Оно начинается с активации вызова вида подмнож $(s1, s2)$, где $s2$ — терм, не содержащий переменных. Допустим, что в ответ на него вызывается процедура Р1. В этом случае следующим активируется вызов процедуры пустое и мы предполагаем, что эта встроенная процедура всегда обрабатывает подобный вызов за конечное время. Если же вызывается процедура Р2, то тогда следующий активируемый вызов имеет вид $t \in s2$.

Посмотрим, как этот вызов \in обрабатывается. Если $s2$ не является структурированным термом $v : x'$, то данный вызов за конечное время приведет к неудаче, поскольку ни процедура Р3, ни процедура Р4 на него не отвечают. В противном случае $s2$ есть некоторый не содержащий переменных терм $r : s2'$. Если вызывается процедура Р3, то наш вызов решается немедленно.

Если же вызывается Р4, то затем активируется вызов $t \in s2'$, второй аргумент $s2'$ которого *меньше*, чем $s2$; очевидно, что последующие обращения к процедуре Р4 для обработки вызова ϵ не могут повторяться бесконечное число раз, поскольку второй аргумент в рекурсивных вызовах ϵ не может бесконечно уменьшаться в размерах. Таким образом устанавливается, что каждый вызов ϵ должен привести к успеху или неудаче за конечное время.

Теперь еще раз обратим свое внимание на вызов $t \in s2$. Если он оказывается неудачным, то неудачным является и вычисление Г. Если же этот вызов решается успешно, то очередной активируемый вызов имеет вид подмнож $(s1, s2')$, где терм $s2'$ *меньше*, чем $s2$. С помощью таких же рассуждений, как и выше, можно показать, что рекурсивные вызовы процедуры Р2 не могут продолжаться бесконечно долго. Итак, мы доказали, что все вызовы процедур пустое, ϵ и подмнож должны обрабатываться за конечное время. Отсюда вытекает, что вычисление Г обязано быть конечным. Этот вывод имеет место для всех вычислений Г, и, стало быть, полное пространство вычислений нашей программы конечно. Следовательно, исполнение программы 29 при стандартной стратегии управления или при любой другой исчерпывающей стратегии завершается. Доказательство завершаемости можно без труда провести более формально с помощью структурной индукции.

V.8. Ограничения на верификацию

Показанный только что метод верификации программ не всегда может привести к успеху. Во-первых, рассматриваемая программа или алгоритм могли бы и не удовлетворять достаточным условиям, если бы, к примеру, множество процедур было неполным или же не все его утверждения логически следовали из спецификации. В этом случае можно было бы надеяться доказать правильность только с помощью необходимых условий, которые потребовали бы тщательного рассмотрения конкретного целевого утверждения и конкретной стратегии управления.

Более фундаментальные препятствия для верификации программ формулируются в различных ограничительных теоремах, давно уже установленных для формальных логических систем. Например, полуразрешимость проблемы общезначимости в логике первого порядка (см. гл. I) означает, что не существует никакой надежной процедуры, позволяющей решать, является или нет каждая данная программа частично правильной, полной или полностью правильной. Таким образом, на возможности верификаторов программ сразу накладывается теоретическое

ограничение. В силу тех же фундаментальных причин невозможно построить процедуры, которые позволяли бы полностью распознавать другие свойства, такие как разрешимость программ или непротиворечивость спецификаций.

Точно так же невозможно придумать никакого надежного теста, позволяющего определять завершаемость логических алгоритмов. Возможности исследования этого вопроса с помощью каких-либо методов доказательства теорем, основанных на характеризации конечности пространства вычислений, могут быть ограничены в зависимости от того, как сформулированы интересующие нас свойства, пределами разрешимости логики либо первого, либо второго порядка. То же самое подтверждается и тем фактом, что каждый логический алгоритм может быть представлен некоторой машиной Тьюринга, ибо, как хорошо известно, вопрос о завершаемости таких алгоритмов («проблема остановки») является только полуразрешимым.

Указанные теоретические препятствия не должны, конечно, удерживать нас от попыток построить автоматические и полуавтоматические средства верификации логических программ, поскольку для подавляющего большинства практически действующих программ вполне возможно доказать их правильность (или неправильность) с помощью систематических и эффективных аналитических методов. Помимо этого, важно также осознавать, что приведенные здесь критерии правильности можно рассматривать как условия, управляющие *синтезом* необходимо правильных программ. Использование проверенных синтезирующих систем, основанных на этих критериях, позволяет эффективным образом избавиться от рассмотрения вообще всех неправильных программ, и тем самым мы в значительной степени преодолеваем ограничения разрешимости, которые потенциально оказывают влияние на логический анализ произвольно построенных программ. Более подробно вопросы синтеза программ рассматриваются в следующей главе.

V.9. Исторический очерк

Первые формулировки критериев верификации принадлежат Кларку и Тернлунду (1977). Они рассматривали верификацию лишь как один из вопросов в общей первопорядковой теории программ и данных. Эта теория развивается и систематизируется в диссертации Кларка (1979), где она описывается как «теория вычисляемого программой отношения». Мы будем называть ее здесь для краткости теорией ВПО.

«Вычисляемое программой отношение», обозначаемое здесь через \tilde{R} , определяется следующим образом:

$$\tilde{R} = \{T \mid P \models R^*(T)\}$$

где P — множество процедур, а $G : ?R^*(t)$ — целевое утверждение. Таким образом, отношение \tilde{R} — это просто множество всех вычислимых с помощью P решений, покрывающее все возможные выборы целевого кортежа t . От целевого утверждения оно, следовательно, не зависит. (То, что мы называли в этой главе «вычисляемым отношением» R , образует подмножество отношения \tilde{R} и состоит из решений, которые вычисляются для заданного целевого утверждения G . Указанное различие между анализируемыми отношениями отражает тот факт, что Кларк предпочитает считать «программой» не пару (P, G) , а лишь множество процедур P ; каждое целевое утверждение G рассматривается в этом случае как «применение» программы P .)

Анализ Кларка и Тернлунда основывается на рассмотрении процедур из P как первопорядковых аксиом, описывающих \tilde{R} . Эти процедуры составляют главную часть множества аксиом A , из которого может быть развита теория отношения \tilde{R} , позволяющая устанавливать различные свойства R , а следовательно, и программы P . Теория ВПО состоит из множества аксиом A и всех предложений (теорем), доказуемых из A в исчислении предикатов первого порядка. В зависимости от исследуемых свойств может потребоваться включить в множество A аксиомы, делающие теорию более сильной, чем если бы A содержало одни лишь процедуры из P ; обычно в этих дополнительных аксиомах кодируются разнообразные правила индукции.

Верификацию программ с помощью теории ВПО можно осуществлять, доказывая из A теоремы вида

$$s : \text{для всех } T \ R^*(T) \Leftrightarrow D(T)$$

где $D(T)$ — некоторое очевидно правильное определяющее для $R^*(T)$. Фактически мы выводим из утверждений программы желаемую спецификацию.

Кроме того, Кларк (1979) представил верификацию логических программ в стиле доказательства правильности традиционных программ, включив в условия верификации входные и выходные предикаты. Так, например, критерий полной правильности может быть представлен в теории ВПО в качестве требования, согласно которому некоторое предложение вида

$$s' : \text{для всех } T_1 \text{ если } I(T_1)$$

$$\text{то для всех } T_2 \ R^*(T_1, T_2) \Leftrightarrow O(T_1, T_2)$$

должно быть доказуемо из A . В приведенной формулировке T_1 и T_2 обозначают некоторые выбранные подмножества аргументов отношения R^* . $I(T_1)$ — это некоторый предикат, описывающий предполагаемые входные аргументы T_1 , тогда как предикат $O(T_1, T_2)$ описывает желаемую зависимость между выходом T_2 и входом T_1 . В предложении s' как раз и требуется,

чтобы получаемые с помощью программы решения вызова R^* в точности удовлетворяли отношению \mathbf{O} между входом и выходом всякий раз, когда на входных данных выполняется предикат I .

Теорию ВПО можно использовать также для доказательства многих других свойств логических программ, например их разрешимости или эквивалентности каким-то иным программам. Предложенный Кларком для достижения этих целей подход полностью аналогичен подходу, который обычно применяется для анализа программ, написанных на других языках, и основывается, в частности, на различных видах индукции. Верификация программ посредством вывода их процедур была исследована вскоре после создания теории ВПО, и, по-видимому, это дает более простой метод доказательства частичной правильности. Сама возможность формализации большей части теории ВПО в логике первого порядка представляется важной для автоматизации обработки данных на метауровне.

В литературе по верификации логических программ имеется некоторая терминологическая путаница, что может вызвать недоразумение у тех, кто не знаком с происходившими в процессе развития этой теории изменениями в определениях. В подходе Кларка и Тернлунда принимается стандартное понятие частичной правильности, однако, пытаясь достичь согласованности с понятиями из области доказательства правильности традиционных программ, в качестве дополнительного требования для определения полной правильности они выбирают «завершаемость». Требование полноты логических программ в явном виде в этой формулировке не рассматривалось. Для исследования полной правильности с помощью теории ВПО им потребовалось поэтому определять завершаемость таким образом, чтобы ее можно было доказывать в исчислении предикатов первого порядка. Это было достигнуто за счет определения завершаемости как свойства, которое в данной книге называется разрешимостью — т. е. существование по крайней мере одного успешно завершающегося вычисления. Точно такой же подход несколькими годами раньше был предпринят Ченом и Ли (1973) при использовании логики дизъюнктов для верификации традиционных программ.

Разрешимость, конечно же, не имеет никакого отношения к обычному понятию завершаемости, которое указывает на окончание всего процесса исполнения программы за конечное время. Исключение составляет тот особый случай, когда исполнение оказывается к тому же детерминированным (т. е. дающим в точности одно вычисление). Именно возможная недетерминированность логических программ лежит в основе невозможности их верификации в точном соответствии с методами доказательства правильности традиционных программ. Причины этого заключаются в том, что (а) потенциальное наличие целого мно-

жества решений предполагает введение требования полноты, (б) независимость логики от управления предполагает существование различий между верификацией программ и верификацией алгоритмов и (с) потенциальное наличие целого множества вычислений и многообразие возможных стратегий управления затрудняют как точное определение завершаемости, так и ее анализ. До сих пор, однако, предпринималось слишком мало исследований, направленных на разработку систематических методов доказательства завершаемости логических алгоритмов.

Верификация с помощью вывода процедур была разработана независимо Кларком (1977), Кларком и Сикелем (1977) и Хоггером (1977, 1978а). Кларк ссылался на нее как на «верификацию следования», подчеркивая тем самым, что ее цель состоит в доказательстве того, что процедуры программы являются логическими следствиями ее спецификации. Более поздние формулировки даются Кларком (1979) и Хоггером (1981, 1982а). Рассматриваемый как средство либо верификации, либо синтеза программ вывод логических процедур отличается от анализа традиционных программ в том, что (а) он позволяет избежать решения обременительной задачи, связанной с аксиоматизацией разнородных и сильно зависящих друг от друга машинно-ориентированных программистских конструкций; (б) он позволяет сосредоточить внимание только на логических свойствах алгоритмов, игнорируя все особенности их управления и (с) он не зависит от целевого утверждения и, следовательно, является нейтральным по отношению к предполагаемому использованию выводимых процедур. Кроме того, концептуальный базис этого метода прост и интуитивно понятен.

Несмотря на все эти преимущества, возникают, однако, и практические трудности: довольно часто между спецификацией и целевыми процедурами не существует логической близости в том смысле, что построение требуемых выводов представляет собой непростое упражнение в доказательстве теорем. Чаще всего это случается тогда, когда в логике программы учитываются очень тонкие свойства проблемной области, как это может быть во многих математических и естественно-научных приложениях. На сложность доказательства подобных свойств выбор формальной системы программирования значительного влияния не оказывает.

С другой стороны, в большом числе практически используемых программ (особенно в тех, которые связаны с коммерческой обработкой данных) рассматриваются лишь сравнительно тривиальные свойства специфицируемых отношений. Именно в такого рода приложениях использование логики должно дать значительные преимущества. Они достигаются не только за счет той ясности, которую логика придает самим программам, но

также благодаря тому, что даже в случае очень больших программ их верификация с помощью вывода процедур будет включать в себя лишь соответствующее большое число триивиальных доказательств. Для подобных программ, по всей видимости, можно разработать системы автоматического поиска вывода, действующие, быть может, во взаимодействии с пользователем. Такого рода реализация была описана Балогом (1981), который использовал правила вывода, предложенные Хоаром (1969). Другие реализации предлагались Ханссоном и Тернлундом (1979), а также Ханссоном и Иоханссоном (1980). Обе они основаны на системах натурального вывода. Хорошие введения в натуральный вывод имеются у Куайна (1959) и Манны (1974).

Наконец, вывод логических процедур был применен также Кларком и ван Эйденом (1981) к доказательству правильности традиционных программ. Они показали, как можно выводить логические процедуры, представляющие логическое содержание блок-схем программ, а затем верифицировать последние посредством доказательства того, что найденные процедуры удовлетворяют заданным спецификациям. Они установили также интересные взаимосвязи между их подходом и различными традиционными методами верификации.

VI. Формальный синтез программ

В этой главе рассматриваются составление и модификация логических программ. Пока еще нельзя предложить действительно систематических методов разработки программ, поскольку у нас нет ни простой теории, позволяющей охарактеризовать эффективные алгоритмы, ни практических способов нахождения таких алгоритмов. Тем не менее представляется полезным очертить логические рамки для вывода программ, на которые могли бы быть наложены механические правила действий, как только наши знания о программах и алгоритмах будут улучшены. Для представления этих рамок мы воспользуемся описанными в гл. V понятиями логической спецификации и логического вывода процедур. Мы предполагаем также, что читатель имеет некоторый опыт работы с выводами в логике первого порядка.

VI.1. Правильность программ

Важным соображением, возникающим при разработке любой программы, является вопрос о том, как предполагается и предполагается ли вообще гарантировать ее правильность. Любое решение этого вопроса должно повлиять на стиль и дух, в которых мыслится и воспроизводится на языке программирования логика программы. Оно, в частности, будет определять роль, играемую той спецификацией, которой должна соответствовать окончательная программа.

Справедливости ради стоит, вероятно, сказать, что «типичный» программист предпочитает разрабатывать программу интуитивным и экспериментальным образом. При составлении утверждений программы он полагается, прежде всего, на свое мысленное впечатление о некотором воображаемом алгоритме. И только потом он может обращаться к точной спецификации с целью оценки правильности уже полученной программы. Такой аналитический подход можно приблизительно описать как «сначала эксперимент, затем анализ». Эти действия будут, возможно, несколько раз итерироваться с тем, чтобы внести исправления и усовершенствования. Приоритет в аналитическом подходе

отдается задаче описания известного эффективного алгоритма, причем всевозможные логические дефекты в этом описании устраняются позднее посредством анализа соответствия программы ее спецификации. Это, быть может, не самый продуктивный или надежный способ программирования, и тем не менее именно его склонны применять большинство программистов, если, конечно, они не ограничены какой-либо навязанной им методологией.

Можно воспользоваться другим подходом, который в отношении вопроса правильности программ является не аналитическим, а *синтетическим*. Шаги программы, согласно этому подходу, выводятся логически из спецификации. Тем самым будет гарантирована их логическая правильность, несмотря на то что всякий выбор между альтернативными шагами в выводе будет определяться исходя из рассмотрения поведения программы в период ее исполнения. Приоритет здесь отдается получению первоначальной версии программы, которая с необходимостью дает правильные решения, в то время как усовершенствования, касающиеся эффективности, могут быть сделаны потом с помощью сохраняющих правильность преобразований. Спецификация играет теперь активную, определяющую роль на протяжении всего процесса создания программы. Она вынуждает каждый вновь создаваемый шаг выполнять некоторое логически необходимое требование, и, таким образом, общая цель постоянно находится в центре внимания программиста. В современной методологии программирования отдается предпочтение именно этому подходу, требующему поддержания логической целостности программы с самого начала ее разработки.

VI.2. Синтез логических программ

Имеется несколько причин, в силу которых синтетический подход оказывается особенно удобным для разработки логических программ. Во-первых, метод верификации, описанный в гл. V как «вывод процедур», сразу же дает нам средство синтеза, способное гарантировать соответствие программ своим спецификациям: в процессе построения вывода эффективным образом получается текст программы. Во-вторых, отделение логики от управления означает, что программист может пользоваться этим средством, заранее не беспокоясь о всех деталях поведения: хотя синтез каждого сегмента программы может и предполагать некоторое определенное поведение, последнее самим сегментом окончательно не фиксируется, поскольку имеется возможность выбора того или иного управления независимо от логики сегмента. В-третьих, записывая спецификации и про-

граммы на одном и том же языке логики первого порядка и связывая их через простое понятие логического следования, программист может непосредственно наблюдать, какой вклад делают те или иные допущения, входящие в спецификацию, в вычислительное решение задачи. По существу синтез программы заключается в выборе полезных с точки зрения вычисления фактов из полной базы знаний, задаваемой спецификацией. Выражая эти факты в виде утверждений логической программы и применяя к ним процедурную интерпретацию, программист может исследовать, какой операционный вклад в алгоритм они делают при различных способах управления. Читая же их чисто декларативно он может увидеть, что они говорят о самой задаче.

Традиционные программистские формализмы этими достоинствами логики не обладают. Прежде всего, из большинства стандартных методов верификации программ нельзя тривиальным обращением получить синтетические средства для порождения алгоритмических конструкций, реализующих заданные логические цели. Более того, синтез традиционных программ не дает возможности откладывать принятие решений относительно их поведения в период исполнения. В частности, использование упорядоченных последовательностей деструктивных присваиваний приводит к сильной логической и операционной зависимости между сегментами программы, что затрудняет их независимую разработку и анализ, делает их менее гибкими при окончательном применении. Наконец, семантическое несоответствие между декларативными, основанными на логике спецификациями и операционными конструкциями машинно-ориентированных языков заметно уменьшает понимание программистом их взаимосвязи, зависящей фактически от большого и разнородного множества соотношений, которые связывают эти два вида формальных систем.

В приводимых ниже иллюстрациях синтеза логических программ мы принимаем в качестве стандартного метода синтеза вывод процедур: каждая процедура программы логически выводится из заданной полной, непротиворечивой, очевидно правильной спецификации, целиком записанной на языке логики первого порядка. Управление процессом построения вывода основывается на соображениях алгоритмической полезности предпринимаемых шагов, которая оценивается неформально.

VI.3. Синтез программ при помощи вывода процедур

В нашем рассмотрении практической верификации программ в разд. V.7 предыдущей главы была предложена идея построения логического вывода из спецификации S единственного

предложения, дающего множество процедур для того или иного конкретного отношения. Например, из спецификации, описывающей множества и подмножества, мы вывели предложение

$$\text{подмнож}(x, y) \Leftrightarrow \text{пустое}(x) \vee (\exists v \exists x')(x = v : x', v \in y, \text{подмнож}(x', y))$$

из которого следуют две процедуры:

$$\begin{aligned} \text{подмнож}(x, y) &\text{ если } \text{пустое}(x) \\ \text{подмнож}(x, y) &\text{ если } x = v : x', v \in y, \text{подмнож}(x', y) \end{aligned}$$

для отношения подмнож.

В первом из приведенных выше предложений справа от связки \Leftrightarrow стоит дизъюнкция двух случаев, для которых отношение подмнож(x, y) имеет место, причем каждый из этих дизъюнктивных членов образует тело выводимой процедуры подмнож. Такого рода предложения могут стать очень громоздкими, если потребуется рассматривать достаточно много случаев. Вместо этого можно воспользоваться более компактным методом, при котором каждая процедура выводится из спецификаций S отдельно, т. е. случаи рассматриваются по одному.

Удобно соединить этот метод с еще одной новой возможностью, которую мы назовем *целенаправленным выводом*. Под целенаправленным выводом мы понимаем вывод процедуры путем обработки исходного целевого утверждения, состоящего из вызова интересующего нас отношения. Рассмотрим, к примеру, вывод, состоящий из следующей последовательности целевых утверждений:

$$G_1 : ? \text{подмнож}(x, y)$$

⋮

$$G_n : ? \text{пустое}(x)$$

Предположим также, что для каждого $t \geq 1$ мы имеем $S, G_t \vdash \vdash G_{t+1}$. Другими словами, каждое выводимое целевое утверждение логически следует из S и предыдущего целевого утверждения. Если это так, то отсюда вытекает, что процедура

$$\text{подмнож}(x, y) \text{ если } \text{пустое}(x)$$

логически следует из S; заголовком процедуры является вызов из G_1 , а ее телом — множество вызовов из G_n .

В более общем случае целенаправленный вывод процедуры для какого-либо отношения r начинается с целевого утверждения

$$G_1 : ? r()$$

и заканчивается некоторым целевым утверждением вида

$$G_n : ? r_1(), r_2(), \dots, r_k()$$

Этим выводом устанавливается, что при условии справедливости спецификации S вызов $r()$ можно решить, решая конъюнкцию вызовов $r_1(), \dots, r_k()$. Имея это, конечно, и утверждается в выводимой процедуре

$$r() \text{ если } r_1(), \dots, r_k()$$

К указанному заключению можно прийти более формально, замечая, что если каждое целевое утверждение логически следует из S и предшествующего целевого утверждения, то в силу транзитивности отношения \vdash имеет место логическое следствие $S, G_1 \vdash G_n$. Поскольку каждое целевое утверждение представляет собой формулу, находящуюся под отрицанием, последнее отношение можно переписать в виде

$$S, \neg \text{заголовок} \vdash \neg \text{тело}$$

и тогда отношение

$$S \vdash (\text{заголовок если тело})$$

получается из него как следствие логической метатеоремы, известной под названием теоремы дедукции¹⁾.

Заметим, что последовательность (G_1, \dots, G_n) напоминает обычное вычисление сверху вниз из логической программы. Отличие состоит только в том, что целевые утверждения, расположенные между G_1 и G_n , не обязаны быть лишь конъюнкциями предикатов. Например, вторым целевым утверждением в этой последовательности могло бы быть такое

$$G_2 : ? (\forall u)(u \in y \text{ если } u \in x)$$

Целевые утверждения образуются здесь в таком же духе, как и при исполнении логической программы. Вследствие использования резолюции исполнение программы (P, G_1) порождает каждое новое целевое утверждение G_{t+1} так, чтобы выполнялось отношение $P, G_t \vdash G_{t+1}$. При выводе процедур с целью выбора информации на каждом шаге вместо множества процедур P используется спецификация S , и в общем случае для работы с более произвольным синтаксисом целевых утверждений требуются более сложные правила вывода, чем правило резолюции.

Преимущество представления вывода процедур в этой квазивычислительной форме состоит не только в большей компактности, достигаемой благодаря тому, что одновременно ищется

¹⁾ Здесь можно воспользоваться определением логического следствия, согласно которому $S \vdash F_1 \vdash \neg F_2$ эквивалентно $S \vdash \neg F_2 \text{ если } \neg F_1$, и законом контрапозиции $\neg F_2 \text{ если } \neg F_1 \vdash F_1 \text{ если } F_2$ — Прим. перев.

вывод только одной процедуры, но также в точном выявлении того, как именно спецификация способствует решению задачи, поставленной изначально в виде целевого утверждения $\exists t \ldots$. Эта общая цель синтеза программы поддерживается в мыслях программиста за счет действий, которые ему нужно выполнять на каждом шаге и которые заключаются в применении некоторого выбранного из S факта к текущему целевому утверждению G_t обоснованным с точки зрения вычисления образом. Заметим, что в принципе программист мог бы ввести вывод в ту точку, где действительно решается исходное целевое утверждение, как будто бы он исполнял «программу», «множеством процедур» которой является спецификация S . На практике, однако, этот вывод обычно заканчивается в некоторой более ранней точке, где может быть получена какая-либо полезная процедура. Поэтому, если цель исполнения программы — получить решение, то в процессе построения вывода процедур ищется *способ* (процедура) получения решения.

VI.4. Пример с использованием резолюции

Иногда одна только резолюция может служить в качестве практической системы логического вывода для получения программ из спецификаций. Мы продемонстрируем это на следующем примере, в котором рассматривается задача нахождения всех пар (u, v) элементов множества z , удовлетворяющих некоторому интересующему нас бинарному отношению $p(u, v)$. Например, если $z = \{1, 2, 3\}$, а p есть отношение «больше» ($>$), то решениями будут пары $(2, 1)$, $(3, 1)$ и $(3, 2)$.

Желаемую зависимость между u , v и z можно описать с помощью предиката $\text{nайти}(u, v, z)$. Отношения, нужные для решения этой задачи, можно специфицировать тогда следующими определениями:

$$\begin{aligned} \text{nайти}(u, v, z) &\Leftrightarrow u \in z, v \in z, p(u, v) \\ u \in z &\Leftrightarrow (\exists w \exists z')(z = w : z', (u = w \vee u \in z')) \end{aligned}$$

и полным определением предиката $=$,

Поскольку для синтеза программы мы намерены использовать резолюцию, удобно извлечь из этих предложений ряд три-вальных логических следствий, каждое из которых имеет структуру стандартной логической процедуры:

- S1 : $\text{nайти}(u, v, z)$ если $u \in z, v \in z, p(u, v)$
- S2 : $u \in z$ если $\text{nайти}(u, v, z)$
- S3 : $v \in z$ если $\text{nайти}(u, v, z)$
- S4 : $p(u, v)$ если $\text{nайти}(u, v, z)$
- S5 : $u \in u : z'$
- S6 : $u \in w : z'$ если $u \in z'$

Предложения S_1-S_4 — это просто отдельные импликации, содержащиеся в приведенном выше определении отношения *найти*, тогда как S_5 и S_6 — стандартные процедуры ϵ , вытекающие из определения отношения ϵ . Все множество $\{S_1, \dots, S_6\}$ может теперь служить в качестве спецификации S для синтеза программы *найти*. Отношение p остается неспецифицированным — любое определение p , такое, скажем, как множество фактов p , может быть добавлено к программе в виде произвольной структуры данных позднее.

Прежде чем двигаться дальше, стоит заметить, что полученную спецификацию можно было бы непосредственно использовать для вычисления решений задачи *найти*. Допустим, к примеру, что в качестве p выбрано (встроенное) отношение $>$, а целевым утверждением является $? \text{ найти}(u, v, I : 2 : 3 : \emptyset)$. Все решения можно вычислить тогда (с довольно сносной эффективностью) только с помощью множества процедур $\{S_1, S_5, S_6\}$. Тем не менее мы приступим к построению другого множества процедур и впоследствии сравним два этих множества.

Мы начнем синтез программы, поставив наиболее общее целевое утверждение

$$G_1 : ? \text{ найти}(u, v, z)$$

так что на класс задач *найти*, решаемых с помощью выводимых процедур, никаких ограничений не накладывается. Применяя резолюцию сверху вниз, можно построить следующий вывод, в котором активированные вызовы подчеркнуты

$$\begin{aligned} G_1 &: ? \underline{\text{найти}}(u, v, z) \\ G_2 &: ? \underline{u \in z}, v \in z, p(u, v) \\ G_3 &: ? \underline{v \in u : z'}, p(u, v) \text{ (присваивая } z := u : z') \\ G_4 &: ? p(u, u) \quad \text{(присваивая } v := u) \end{aligned}$$

Здесь целевое утверждение G_2 получается в результате вызова процедуры S_1 , G_3 — в результате вызова S_5 , и G_4 — также в результате вызова S_5 . Продолжение этого вывода посредством вызова процедуры S_4 в ответ на целевое утверждение G_4 не послужило бы никакой очевидной цели, поскольку это привело бы к появлению вызова *найти*, в котором упоминается некоторое произвольное множество, никак не связанное с рассматриваемым в данный момент множеством z . Поэтому в этой точке мы выводим первую процедуру *найти*

$$(\text{найти}(u, v, z) \text{ если } p(u, u))\Theta$$

где Θ — множество накопленных связываний $\{z := u : z', v := u\}$. Итак, мы получаем процедуру

$P1 : \text{найти}(u, u, u : z') \text{ если } p(u, u)$

Эта процедура пытается решить каждый вызов найти (u, v, z) , выбирая в качестве u и v первый элемент множества z и затем показывая, что справедливо отношение $p(u, u)$.

Очевидно, что есть и другие способы выбора u и v из множества z . Их можно обнаружить с помощью механизма возврата в только что построенном выводе, осуществляя поиск шагов, допускающих альтернативные неиспробованные выборы процедур. Так, если мы вернемся к целевому утверждению G_3 , то обнаружим, что вместо $S5$ можно вызвать процедуру $S6$ и породить тем самым новое целевое утверждение

$G'_4 : ? v \in z', p(u, v) \text{ (присваивая } w := u)$

В этой точке в ответ на вызов ϵ можно было бы обратиться либо к процедуре $S5$, либо к $S6$, но в результате появились бы ссылки на компоненты множества z' ; в рассматриваемом синтезе мы отказываемся от этой возможности и закончим построение вывода целевым утверждением G'_4 для того, чтобы вывести еще одну процедуру найти

$P2 : \text{найти}(u, v, u : z') \text{ если } v \in z', p(u, v)$

Она берет в качестве u первый элемент множества z , v выбирает из оставшейся части z' и затем пытается показать, что имеет место отношение $p(u, v)$.

Точно так же мы можем вернуться к целевому утверждению G_2 и построить следующий вывод:

$G_2 : ? \underline{u \in z}, v \in z, p(u, v)$

$G'_3 : ? \underline{u \in z'}, v \in w : z', p(u, v) \text{ (присваивая } z := w : z')$

$G''_4 : ? \underline{u \in z'}, p(u, v) \text{ (присваивая } w := v)$

Здесь G'_3 получается в результате вызова процедуры $S6$, а G''_4 — в результате вызова $S5$. В этой точке выводится процедура

$P3 : \text{найти}(u, v, v : z') \text{ если } u \in z', p(u, v)$

которая отличается от $P2$ только тем, что при выборе из множества z элементы u и v меняются ролями.

Наконец, можно вернуться к целевому утверждению G'_3 и в ответ на второй вызов ϵ обратиться к процедуре $S6$, получая

вывод

- $G_4''' : ? \underline{u \in z'}, v \in z', p(u, v)$
- $G_5 : ? \underline{\text{найти}(u, v, z')}, v \in z', p(u, v)$ (вызывая S2)
- $G_6 : ? \underline{\text{найти}(u, v, z')}, \underline{\text{найти}(u, v, z')}, p(u, v)$ (вызывая S3)
- $G_7 : ? \underline{\text{найти}(u, v, z')}, \underline{\text{найти}(u, v, z')}, \underline{\text{найти}(u, v, z')}$
(вызывая S4)
- $G_8 : ? \underline{\text{найти}(u, v, z')}$ (склеивая одинаковые вызовы)

Здесь мы вновь воздержались от обращения к процедурам S5 и S6 в ответ на вызовы $\underline{\epsilon}$ с тем, чтобы избежать декомпозиции множества z' . В построенном выводе рассматривается случай, когда и u и v ищутся в оставшейся части z' , что и выражается процедурой, выводимой в этой точке:

$P4 : \text{найти}(u, v, w : z') \text{ если } \text{найти}(u, v, z')$

Полученным множеством процедур $\{P1, \dots, P4\}$ исчерпываются все способы выбора u и v из первичных компонент w и z' множества $z = w : z'$. Эти процедуры вместе с S5, S6 и данными, описывающими отношение p , дают полное множество процедур для исследования любой задачи найти. Например, если бы в качестве данных для p были выбраны факты

$p(2, 1)$
 $p(1, 3)$
 $p(x, x)$

то целевое утверждение $? \text{найти}(u, v, 1 : 2 : 3 : 0)$ решалось бы полностью, и решениями (u, v) были бы пары $(1, 1)$, $(1, 3)$, $(2, 1)$, $(2, 2)$ и $(3, 3)$.

Приведенный только что пример синтеза можно рассматривать как ограниченное исследование определяемого посредством G_1 и S полного дерева выводов сверху вниз. Каждый отдельный путь в этом дереве, ведущий из исходного целевого утверждения G_1 к некоторому заключительному целевому утверждению на границе поиска, дает некоторую выводимую процедуру; это изображено на рис. VI. 1. Данное исследование ограничено с двух сторон: (а) граница поиска может продвигаться только на определенное расстояние вниз по дереву выводов и (б) не выполняются шаги, на которых либо разбираются первичные компоненты множества z , либо появляется целевое утверждение, совпадающее с уже полученным ранее, либо вводятся множества, не имеющие к z никакого отношения. Более позитивно, используемая стратегия ограничивает анализ задачи, поставленной целевым утверждением G_1 , исследованием всех способов выбора u и v из первичных компонент z ; вследствие этого на рис. VI. 1 показаны не все пути в полном дереве выводов, поскольку не

все они были просмотрены. Иные ограничения были бы мотивированы иными представлениями о вычислении и привели бы к другим множествам процедур.

Полноту множества процедур $\{P_1, \dots, P_4\}$ для отношения найти можно неформально доказать на основании того факта, что всякий раз, когда вызывается процедура S_5 в ответ на вызов ϵ должным образом будет вызываться также и процедура S_6 .

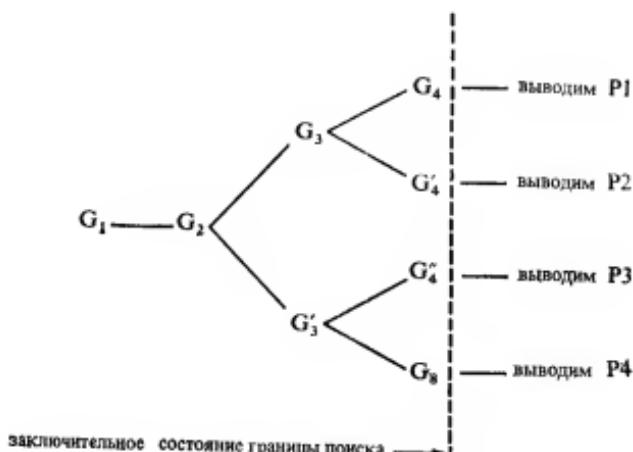


Рис. VI. 1. Процедуры, получающиеся из дерева вывода.

Поведение, даваемое процедурами $\{P_1, \dots, P_4\}$, во многом подобно тому, которое получилось бы в результате использования альтернативного множества $\{S_1, S_5, S_6\}$. Однако первое множество процедур легче приспособить для извлечения преимущества из каких-либо особых свойств, которыми, быть может, обладает выбранное отношение r . Пусть, например, известно, что отношение r иррефлексивно, т. е. не содержит пар вида (u, u) . Тогда процедуру P_1 можно вычеркнуть из программы, улучшая тем самым эффективность ее исполнения, поскольку не будут проводиться безрезультатные поиски пар (u, u) . Точно так же, если известно, что отношение r симметрично, т. е. что оно содержит пары (u, v) и (v, u) одновременно, то либо процедуру P_2 , либо процедуру P_3 (но ни ту и другую вместе) можно вычеркнуть из программы, получая при этом определенные преимущества. С другой стороны, в случае процедур $\{S_1, S_5, S_6\}$ не имеется никакого простого способа изменения логики или управления, обеспечивающего такие же улучшения эффективности. Таким образом, наш синтез может рассматриваться как сохраняющее правильность преобразование процедур $\{S_1,$

$S_5, S_6\}$ в процедуры $\{P_1, \dots, P_4\}$, в результате которого получаются более гибкие средства, позволяющие улучшать поведение программы в период ее исполнения.

VI.5. Пример с использованием нерезолютивного вывода

В принципе резолюция является достаточным средством для вывода логических процедур из спецификаций, поскольку каждое предложение стандартной логики, входящее в нашу исходную спецификацию, можно преобразовать с помощью различных систематических методов в (фактически) эквивалентное множество предложений, представленных в виде дизъюнктов. В зависимости от вида исходного предложения это множество может содержать как хорновские, так и нехорновские дизъюнкты. Если все дизъюнкты в нем хорновские, то в последующем синтезе можно использовать простой метод резолюции сверху вниз, показанный в предыдущем примере. В противном случае (когда имеются нехорновские дизъюнкты) можно применить общую резолюцию.

На практике такой подход почти всегда оказывается неудовлетворительным главным образом потому, что стандартное преобразование спецификации в множество дизъюнктов приводит, как правило, к значительной потере компактности и понятности. Трудно к тому же сопоставить общей резолюции какую-либо непосредственную вычислительную интерпретацию в духе решения задач, способную придать процессу вывода целенаправленность. Чаще поэтому предпочтительнее не преобразовывать спецификацию в множество дизъюнктов, а прямо из нее выводить желаемые процедуры, пользуясь теми правилами вывода, которые представляются наиболее удобными. Платой за это является тот факт, что эти правила зачастую оказываются более сложными и менее единообразными, чем правило резолюции.

В данном разделе мы продемонстрируем использование нерезолютивного вывода на примере применения его к задаче преобразования программ. Пусть имеется исходная программа, работающая с конкретным представлением данных. Цель преобразования — модифицировать программу так, чтобы она могла работать с некоторым другим представлением.

VI.5.1. Исходная программа

Задача программы состоит в том, чтобы проверить, является ли данный список L упорядоченным в соответствии с некоторым отношением порядка $<$. Например, список $L = (3, 5, 7, 9)$ будет упорядоченным в соответствии с обычным отношением $<$.

(«меньше») на множестве целых чисел. Приводимая ниже программа 35 представляет собой первоначальную версию. Процедуры ее специально предназначены для работы со списками, которые представлены обычными структурированными термами, построенными из точек и константы *NIL*.

Программа 35

- ? *поряд*(3.5.7.9.*NIL*)
- P1 : *поряд*(*x*) если *длина*(*x*, *n*), *n* ≤ 1
- P2 : *поряд*(*x*) если *склейт**(*u*, *x'*, *x*), *склейт**(*v*, *x''*, *x'*),
 u < *v*, *поряд*(*x'*)
- P3 : *длина*(*NIL*, 0)
- P4 : *длина*(*w.NIL*, 1)
- P5 : *склейт**(*w*, *z'*, *w.z'*)

Здесь предикат *поряд*(*x*) означает, что список *x* является упорядоченным, *длина*(*x*, *n*) означает, что длина списка *x* равна *n*, а предикат *склейт**(*u*, *x'*, *x*) означает, что список *x* получается в результате присоединения списка *x'* к единичному списку (*u*).

Исполнение программы состоит главным образом из итеративных циклов, управляемых процедурой P2, в каждом из которых с помощью двух вызовов *склейт** извлекается первая пара (*u*, *v*) элементов, некоторого текущего фрагмента списка *L* и затем проверяется справедливость отношения *u* < *v*. Если текущий фрагмент содержит менее двух элементов, то благодаря P1 вычисление завершается успешно. Таким образом, для указанного целевого утверждения исполнение программы подтверждает сначала, что $3 < 5$ в фрагменте (3, 5, 7, 9), затем — что $5 < 7$ в фрагменте (5, 7, 9) и, наконец, что $7 < 9$ в фрагменте (7, 9); заключительным фрагментом будет единичный список (9), который согласно P1 и P4 является упорядоченным. Следовательно, наше целевое утверждение решается успешно.

Программа 35, заметим, устроена так, что процедуры P1 и P2 выражают логические свойства упорядоченных списков независимо от того, каким именно способом они представлены. Напротив, и целевое утверждение, и процедуры P3—P5, предназначенные для осуществления доступа к длинам и элементам фрагментов списка *L*, ограничены выбором конкретного представления списков в виде структурированных термов.

VI.5.2. Изменение представления данных

Посмотрим теперь, что получится, если мы решим представить список $L = (3, 5, 7, 9)$ в виде массива, пользуясь следующим множеством фактов:

$$\begin{array}{lll} \text{э}(3, 1, L) & \text{э}(7, 3, L) & \text{длина}(L, 4) \\ \text{э}(5, 2, L) & \text{э}(9, 4, L) & \end{array}$$

Ясно, что процедуры Р3—Р5 придется заменить некоторыми новыми процедурами доступа, предназначеными для работы с этим представлением, а Р1 и Р2 можно оставить без изменений.

Прежде всего, заметим, что цель процедур Р3—Р5 состоит в извлечении либо длины, либо первых двух элементов из каждого фрагмента списка L , образованного в ходе исполнения программы. Так как эти фрагменты сами являются списками, то они, естественно, в процедурах Р3—Р5 представляются структурированными термами, ибо именно это представление выбрано для программы 35. При выводе новой версии процедур Р3—Р5 нам понадобится какой-то другой способ обозначения фрагментов, соответствующий имеющемуся представлению списка L в виде фактов.

Для этих целей существует одна простая форма записи, в которой используются термы вида $f(x, j)$, где x — имя списка (например, L), а j — номер некоторой позиции в x . Весь терм $f(x, j)$ может обозначать тогда тот фрагмент списка x , который остается после вычеркивания всех элементов из x , занимающих позиции с номерами, меньшими чем j . Таким образом, в соответствии с этой схемой фрагменты $(3, 5, 7, 9)$, $(5, 7, 9)$, $(7, 9)$ и (9) будут обозначаться термами $f(L, 1)$, $f(L, 2)$, $f(L, 3)$ и $f(L, 4)$ соответственно. Отметим, что если $j \geq l$, то для списка x длины $m \geq 0$ фрагмент $f(x, j)$ является единичным списком, когда $j = m$, и пустым списком, когда $j > m$.

Новые процедуры доступа, заменяющие Р3—Р6, должны быть в состоянии извлекать длины и элементы фрагментов, представленных в виде $f(x, j)$. При вызове им будут передаваться термы, подобные $f(L, 2)$, которые содержат в точности такую же информацию (т. е. имя списка (L) и номер позиции (2)), как и информация, заключенная в фактах, описывающих список L . Это дает возможность предположить, что новая форма записи фрагментов хорошо подходит для решения задачи получения доступа к представлению списка L в виде фактов.

VI.5.3. Схема новой программы

На этом этапе читателю, видимо, будет полезно заранее представить себе, как будет выглядеть новая программа. Она приводится ниже под номером 36. Впоследствии, однако, эта программа будет несколько усовершенствована с целью улучшения ее эффективности.

Программа 36

```
? поряд(f(L, 1))
Р1 :    как и прежде
Р2 :    как и прежде
```

- P3' : длина($f(x, j), 0$) если $j \geq 1$, длина(x, m), $j = m + 1$
 P4' : длина($f(x, j), 1$) если $j \geq 1$, длина(x, m), $j = m$
 P5' : склейть*($w, f(x, j + 1), f(x, j)$) если э(w, j, x)
 а также факты, представляющие список L

Новые процедуры Р3'—Р5' служат точно тем же целям, что и соответствующие им процедуры Р3—Р5. Заметим, в частности, что в результате любого обращения к процедурам Р3'—Р5' образуются вызовы вида длина (L, m) или э(w, j, L), на которые можно прямо ответить с помощью имеющихся данных, представленных посредством фактов.

Новая программа ведет себя подобно традиционной итеративной программе, последовательно просматривающей линейный массив L под управлением возрастающего индекса j . Исходное целевое утверждение эффективным образом задает начальное значение j , равное 1, а механизм увеличения j на единицу реализуется процедурой Р5'. Итерация завершится тогда, когда будет обработан цикл, в котором рассматривается случай $j = 4$ (т. е. когда значение j будет равно длине списка L). Ниже дается набросок успешного вычисления:

```
? поряд(f(L, 1))
? склейть*(u, x', f(L, 1)),
  склейть*(v, x'', x'), u < v, поряд(x')
.
.
.
с помощью Р5' вычисляются
x' := f(L, 2), u := 3,
x'' := f(L, 3), v := 5
.
.
.
? 3 < 5, поряд(f(L, 2))
.
.
.
? 5 < 7, поряд(f(L, 3))
.
.
.
? 7 < 9, поряд(f(L, 4))
.
.
.
? длина(f(L, 4), n), n ≤ 1
.
.
.
```

с помощью Р4' вычисляется $n := 1$

⋮

□ задача решена

VI.5.4. Спецификация

Связь между программами 35 и 36 основывается на их общей спецификации S. Для того чтобы получить желаемое преобразование, в рассматриваемом примере потребуется использовать определенные факты из спецификации, которые нельзя было бы так просто вывести только лишь из утверждений программы. Мы приводим поэтому полную спецификацию.

- S1 : $\text{поряд}(z) \Leftrightarrow (\forall uvi)(u < v \text{ если } \exists(u, i, z), \exists(v, i + 1, z))$
- S2 : $\text{склеить}^*(w, z', z) \Leftrightarrow (\forall ui)(\exists(u, i, z) \Leftrightarrow \exists(u, i - 1, z') \vee (u = w, i = 1))$
- S3 : $\text{длина}(z, k) \Leftrightarrow 0 \leq k, (\forall i)(1 \leq i, i \leq k \Leftrightarrow (\exists u) \exists(u, i, z))$
- S4 : $(\exists k) \text{ длина}(z, k)$
- S5 : $(u = v \Leftrightarrow \exists(u, i, z)) \text{ если } \exists(v, i, z)$
- S6 : $\neg \exists(u, i, NIL)$
- S7 : $\exists(u, i, w.z') \Leftrightarrow \exists(u, i - 1, z') \vee (u = w, i = 1)$
- S8 : $\exists(u, i, f(x, j)) \Leftrightarrow \exists(u, i + j - 1, x), 1 \leq i, 1 \leq j$

Предложения S1 и S2 — это просто очевидные определения отношений поряд и склеить*. Предложения S3—S5 являются стандартными аксиомами списков, гарантирующими, что каждый список z имеет неотрицательную длину k и ровно по одному элементу на каждой позиции с номерами от 1 до k . Принадлежность элементов спискам, представленным обычными структурированными термами, описывается предложениями S6 и S7, а принадлежность элементов спискам, представленным с помощью термов $f(x, j)$, описывается предложением S8. Все процедуры программ 35 и 36 можно вывести в логике первого порядка из этого множества спецификаций.

VI.5.5. Эквивалентная подстановка

Эквивалентная подстановка — это одно из правил вывода в логике первого порядка, оказавшееся особенно полезным для манипулирования тем видом предложений, которые обычно участвуют в выводе процедур. Наиболее простая форма этого правила такова. Сначала выбирается некоторое предложение, имеющее структуру

$$F_1 \Leftrightarrow F_2$$

и показывается, что либо оно принадлежит S , либо логически следует из S , либо является теоремой (общезначимым предложением) логики первого порядка; в любом случае обязательно справедливо отношение

$$S \models (F_1 \Leftrightarrow F_2)$$

Пусть F_1 и F_2 — произвольные логические формулы, удовлетворяющие этому условию. Затем из текущего целевого утверждения G_t выбирается некоторая подформула F , такая что для некоторого унифициатора Θ имеет место равенство $F = F_1\Theta$. Наше правило вывода заключается тогда в подстановке F_2 вместо F в целевом утверждении G_t и применении к результату унифициатора Θ . Таким образом, получается следующее целевое утверждение G_{t+1} , для которого справедливо отношение

$$S, G_t \models G_{t+1}$$

Например, предварительное условие $S \models (F_1 \Leftrightarrow C)$ дало бы возможность осуществить следующий шаг вывода:

$$\begin{aligned} G_t : & ?(A \Leftrightarrow B, F), D \\ G_{t+1} : & ?(A \Leftrightarrow B, C)\Theta, D\Theta \end{aligned}$$

при условии, что $F = F_1\Theta$.

Один полезный вариант этого правила, известный как «условная эквивалентная подстановка», начинается с установления несколько более сложного предварительного условия

$$S \models (F_1 \Leftrightarrow F_2) \text{ если } F_3$$

где F_1 , F_2 и F_3 — любые логические формулы. Здесь «эквивалентность» F_1 и F_2 обусловлена теперь формулой F_3 . Снова из целевого утверждения G_t выбирается некоторая формула F , такая что $F = F_1\Theta$. На этот раз шаг вывода заключается в подстановке F_2 вместо F , добавлении F_3 в качестве нового конъюнктивного члена к целевому утверждению и, наконец, применении к результату подстановки Θ . Таким образом получается новое целевое утверждение G_{t+1} , которое вновь удовлетворяет отношению

$$S, G_t \models G_{t+1}$$

Например, предварительное условие $S \models (F_1 \Leftrightarrow C) \text{ если } E$ дало бы возможность осуществить следующий шаг вывода

$$\begin{aligned} G_t : & ?(A \Leftrightarrow B, F), D \\ G_{t+1} : & ?(A \Leftrightarrow B, C)\Theta, D\Theta, E\Theta \end{aligned}$$

при условии, что $F = F_1\Theta$.

В этом кратком описании правил игнорируются некоторые незначительные ограничения, накладываемые на допустимое распределение переменных и кванторов, входящих в G_t , F , F_1 , F_2 и F_3 . Однако для целей нашего примера (да и практически большинства других) эти ограничения не имеют никакого значения, и поэтому детализировать здесь мы их не будем.

Одно полезное свойство сформулированных правил состоит в том, что они применяются независимо как от структуры выбранной из целевого утверждения подформулы F , так и того контекста, в котором F встречается в целевом утверждении: F просто выбирается произвольным образом и заменяется некоторой другой формулой, которая в соответствии с S либо безусловно эквивалентна F (как это было в первом правиле), либо эквивалентна F при некотором дополнительном условии (как во втором правиле); к получаемой в результате этой замены формуле затем применяется унификатор Θ , обеспечивающий должную зависимость нового целевого утверждения от унификации, необходимой для замены F .

Применения этого правила будут обычно управляться предвидением (в какой-то степени) логических и вычислительных возможностей тех процедур, которые мы желаем получить. Теперь мы проиллюстрируем сказанное, продолжив рассмотрение нашего примера преобразования программы.

VI.5.6. Вывод новых процедур

Здесь мы приведем выводы новых процедур Р3' и Р4' для отношения длина. В этих процедурах рассматриваются только такие фрагменты списка, длины которых равны либо 1, либо 0. Вместо того чтобы с самого начала разрабатывать процедуры Р3' и Р4' по отдельности, мы будем строить только один вывод, имеющий дело с фрагментами произвольной длины n , а затем конкретизируем полученный результат, рассмотрев случаи $n=0$ и $n=1$. Ниже мы приведем вывод полностью, а потом объясним, как выполняются его различные шаги.

$G_1 : ? \text{ длина}(f(x, j), n)$

$$G_2 : \exists n, (\forall i)(l \leq i, i \leq n \Leftrightarrow (\exists u) \; e(u, i, f(x, j)))$$

$$G_3 : \quad ? \ 0 \leq n, (\forall i)(I \leq i, i \leq n \Leftrightarrow (\exists u)(\exists(u, i + j - 1, x), \\ I \leq i, i \leq j))$$

$$G_4 : \exists \theta \leq n, (\forall i)(I \leq t, i \leq n \Leftrightarrow (\exists u) \ s(u, i + j - I, x),$$

$$G_5^{\leftarrow}: \exists \theta \leq n, (\forall i)(I \leq i, i \leq n \Leftrightarrow (\exists u) \ \exists(u, i + j - I, x))$$

$$G_6 : \quad ? j \leq m+1, (\forall i')(j \leq i', i' \leq m \Leftrightarrow (\exists u) \; s(u, i', x),$$

$G_7 : ? j \leq m + 1, (\forall i')((i' \leq m \Leftrightarrow (\exists u) \exists(u, i', x)) \text{ если } j \leq i', I \leq j)$

$G_8 : ? j \leq m + 1, 0 \leq m, (\forall i')(I \leq i', i' \leq m \Leftrightarrow (\exists u) \exists(u, i', x)), I \leq j$

$G_9 : ? j \leq m + 1, \text{длина}(x, m), I \leq j$

Из этого вывода получается следующая процедура

$\text{длина}(f(x, j), m - j + 1) \text{ если } j \leq m + 1, \text{длина}(x, m), I \leq j$

В ней формулируется довольно очевидная связь между длиной списка x и длиной его фрагмента $f(x, j)$. Шаги в выводе основаны главным образом на применении правил эквивалентной подстановки (ЭП) и условной эквивалентной подстановки (УЭП). Они объясняются ниже.

Шаг G₁ – G₂ : Применяем правило ЭП с помощью предложений S3, выбирая

$F = \text{длина}(f(x, j), n)$

$F_1 = \text{длина}(z, k)$

$F_2 = \text{определяющее, стоящее справа от связки} \Leftrightarrow \text{в S3}$

$\Theta = \{z := f(x, j), k := n\}$

Шаг G₂ – G₃ : Применяем правило ЭП с помощью предложения S8, заменяя $F = \exists(u, i, f(x, j))$

Шаг G₃ – G₄ : Применяем правило ЭП с помощью теоремы $(\exists u)(A, B) \Leftrightarrow (\exists u)A, B$

где формула B не содержит вхождений u .

Применяем правило УЭП с помощью теоремы $((A \Leftrightarrow B, C) \Leftrightarrow (A \Leftrightarrow B))$ если C

в качестве C здесь берем формулу $I \leq j$.

Шаг G₄ – G₅ : Выполняем подстановки $i := i' - j + 1$ и $n := m - j + 1$ и, исходя из этого, заменяем $0 \leq n$ на $j \leq m + 1$

$I \leq i$ на $j \leq i'$

$i \leq n$ на $i' \leq m$

$\exists(u, i + j - 1, x)$ на $\exists(u, i', x)$.

Шаг G₅ – G₆ : Применяем правило ЭП с помощью теоремы $(A, B \Leftrightarrow A, C) \Leftrightarrow ((B \Leftrightarrow C) \text{ если } A)$

в качестве A здесь берем формулу $j \leq i'$.

Шаг G₆ – G₇ : Это наиболее трудный шаг; используем импликацию

$((B \Leftrightarrow C) \text{ если } A) \text{ если } (D, B \Leftrightarrow C), E$

где

A есть $j \leq i'$

B есть $i' \leq m$

C есть $(\exists u) \exists(u, i', x)$

D есть $I \leq i'$

E есть $I \leq j$

Это предложение доказуемо из S с помощью несколько утомительного анализа свойств отношения \leq . И наконец, в целевое утверждение можно ввести дополнительный вызов $0 \leq m$, поскольку он является следствием имеющихся вызовов $1 \leq j$ и $j \leq m+1$, удовлетворяющих свойствам арифметики, которые, как предполагается, вытекают из спецификации S .

Шаг G₈ — G₉ : Применяем правило ЭП с помощью предложе-
ния S3, заменяя определяющее

$$F = 0 \leq m, (\forall i')(1 \leq i', i' \leq m \Leftrightarrow (\exists u) \in(u, i', x))$$

на его определяемое $F_2\Theta = \text{длина}(x, m)$.

Желаемые целевые процедуры можно теперь получить, выбирая по очереди $n = 0$ и $n = 1$. В первом случае имеем $m = j + 1 = 0$ и, следовательно, $j \leq m + 1$ эквивалентно $j = m + 1$. Поэтому выводимую процедуру можно в этом случае представить в виде

$$\text{длина}(f(x, j), 0) \text{ если } 1 \leq j, \text{длина}(x, m), j \leq m + 1$$

Мы получаем, стало быть, процедуру P3'. Во втором случае полагаем $m - j + 1 = n = 1$, что делает формулу $j \leq m + 1$ эквивалентной $j = m$. Мы получаем тогда процедуру

$$\text{длина}(f(x, j), 1) \text{ если } 1 \leq j, \text{длина}(x, m), j = m$$

как раз совпадающую с P4'.

Руководящим принципом при построении этого вывода служило то обстоятельство, что целевая процедура, которую мы стремились получить, должна была иметь возможность вычислять длину n любого фрагмента $f(L, j)$ непосредственно из фактов, содержащихся в базе данных. В эту базу данных входит ряд фактов о принадлежности элементов списку и о длине m списка L ; по-видимому, только последний факт может сыграть какую-либо полезную роль в определении n . Поэтому и весь вывод строился таким образом, чтобы получить в заключительном целевом утверждении ссылку на предикат $\text{длина}(x, m)$. Это делалось с помощью применения серии подстановочных преобразований к определяющему предиката $\text{длина}(f(x, j), n)$ до тех пор, пока в целевом утверждении G₈ не появилось определяющее предиката $\text{длина}(x, m)$.

Вывод новой процедуры P5' для отношения склеить* значительно проще, чем только что рассмотренный. Он оставляется читателю в качестве упражнения.

VI.5.7. Дальнейшие усовершенствования программы

У программы 36 имеется несколько операционных дефектов. Прежде всего, значительные расходы в период исполнения возникают из-за использования явного интерфейса P3'—P5' доступа

к данным, расположенного между основными процедурами P1—P2 и базой данных. Эффективность исполнения нашей программы можно улучшить, применяя к ее процедурам еще одно простое преобразование, известное как *макрообработка*. В общем случае это преобразование позволяет заменить пару процедур вида

$$\begin{aligned} & \text{A если } B, C, D \\ & \text{C если } E, F, G \end{aligned}$$

одной процедурой

$$\text{A если } B, E, F, G, D$$

устраняя, таким образом, промежуточный вызов **C**. Такого рода шаги могут выполняться до начала исполнения программы автоматически с помощью резолюции посредством обычного обращения к процедуре **C** в ответ на вызов **C**.

Применяя этот принцип к программе 36, можно получить следующую более компактную и эффективную программу.

Программа 37

$$\begin{aligned} & ? \text{ поряд}(f(L, I)) \\ P1' : & \text{ поряд}(f(x, j)) \text{ если } j \geq 1, \text{длина}(x, m), j = m + 1 \\ P1'' : & \text{ поряд}(f(x, j)) \text{ если } j \geq 1, \text{длина}(x, m), j = m \\ P2' : & \text{ поряд}(f(x, j)) \text{ если } \exists(u, j, x), \exists(v, j + 1, x), \\ & u < v, \text{поряд}(f(x, j + 1)) \\ & \text{и база данных, описывающих список } L \end{aligned}$$

Здесь мы обращались по очереди к процедурам P3' и P4' в ответ на вызов **длина** из P1 для того, чтобы получить процедуры P1' и P1'' соответственно. Мы обращались также к процедуре P5' в ответ на вызов **склеить*** из P2 для получения процедуры P2'. Тем самым можно избавиться от интерфейса P3'—P5'.

Нам осталось исправить еще несколько дефектов эффективности. Каждое из правил условной эквивалентной подстановки, применявшихся в выводе целевых утверждений G₅ и G₈, вводит условие $I \leq j$ (т. е. $j \leq I$) в качестве дополнительного вызова в текущее целевое утверждение, а, стало быть, посредством процедур P3' и P4', и в самые последние процедуры P1' и P1''. Последствия выбора альтернативного условия $j < I$ не были исследованы, поскольку нужный нам алгоритм обрабатывает только фрагменты $f(L, j)$ при $j \geq 1$. Несмотря на то что вызовы $j \geq 1$ логически необходимы, они служат помехой эффективным вычислениям, потому что целевое утверждение программы с самого начала инициирует выбор $j = 1$, и в дальнейшем значение j может только возрастать. Процедуры, однако, об этом ничего «не знают» и потому должны осуществлять излишнюю проверку $j \geq 1$ при всяком обращении к ним.

Еще один источник неэффективности программы 37 — это бремя, связанное с унификацией структурированных термов вида $f(x, j)$ в период ее исполнения. Функторы f играли полезную роль для точного описания логического базиса нашего преобразования, но совершение не нужны для окончательной цели вычислений в выведении программе.

Оба этих дефекта можно исправить, добавив к спецификации S предложение

S9 : $\text{поряд}^*(x, j) \Leftrightarrow (\text{поряд}(f(x, j)) \text{ если } j \geq 1)$

и затем переписав программу 37 в терминах предиката **поряд***, а не **поряд**. В результате будут устраниены все функторы f , а также проверки $j \geq 1$, и мы получим следующую окончательную версию программы.

Программа 38

```
? поряд*(L, I)
  поряд*(x, j) если длина(x, m), j = m + 1
  поряд*(x, j) если длина(x, m), j = m
  поряд*(x, j) если э(u, j, x), э(v, j + 1, x),
    u < v, поряд*(x, j + 1)
  и база данных, описывающих список L
```

Используя нерезолютивный вывод, довольно легко показать (для читателя это будет полезным упражнением), что каждое утверждение программы 38 логически следует из своего двойника в программе 37 и расширенной указанным выше образом спецификации S.

VI. 6. Исторический очерк

История синтеза логических программ практически совпадает с историей их верификации, изложенной в конце предыдущей главы. Как мы уже видели, оба этих направления можно рассматривать в одних и тех же рамках, основываясь на выводе процедур, посредством которого устанавливается логическая связь программ с их спецификациями.

Поиски эффективной методологии для построения логических программ первоначально мотивировались проводившимися в начале 70-х годов более общими разработками в области «структурного программирования» для традиционных формализмов. Они привели автора (Хоггер, 1975) к изучению различных логических формулировок популярной тогда «проблемы восьми ферзей» и установлению взаимосвязей между ними. Тем време-

нем Барстолл и Дарлингтон (1977) предложили элегантный автоматический метод разработки программ, написанный на языке рекурсивных равенств, который во многом подобен языку логики хорновских дизъюнктов. Найденные ими правила вывода были впервые приспособлены для синтеза логических программ Кларком и Сикелем (1977).

Методологию Барстолла и Дарлингтона к синтезу семейства нетривиальных алгоритмов из их общей спецификации, представленной на языке логики первого порядка, впервые применил Хоггер (1977), рассмотревший ряд алгоритмов сортировки. Дальнейшие исследования вывода этих алгоритмов проводились Кларком и Дарлингтоном (1980). Использование первопорядкового вывода с целью получения других семейств логических алгоритмов иллюстрировалось Кларком (1977) для задач вычисления факториала и чисел Фибоначчи, Хоггером (1979б) для задачи поиска подстроки, Ханссоном и Тарнлундом (1979) для обработки списков и деревьев, Унтерстайном и др. (1980) для алгоритмов унификации, Маккиманом и Сикелем (1980) для проблемы Хоара FIND, а также Кларком, Маккиманом и Сикелем (1982а) для задачи численного интегрирования. Более подробное изложение теоретического базиса синтеза логических программ можно найти в работах Кларка (1979) и Хоггера (1979а, 1981). Схемы синтеза, подобные тем, которые описаны в этой главе, были независимо обнаружены и детально разработаны Манной и Уолдингером (1980).

В настоящее время не существует полностью автоматической реализации, способной строить выводы эффективных логических программ из произвольных спецификаций, и по всей видимости в ближайшем будущем они не появятся. Тем не менее имеется ряд реализаций, которые могут удовлетворительно работать во взаимодействии с пользователем, причем они способны не только проверять собственные выводы пользователя, но и сами проявлять некоторую инициативу, направленную на построение выводов. В такого рода реализациях, обсуждаемых Ханссоном и Тарнлундом (1979), используется, как правило, система построения натурального вывода, и поэтому они могут обрабатывать спецификации, представленные в неограниченном языке логики первого порядка. Пока у нас еще нет достаточно полного представления о том, как связана общая проблема эффективного управления синтезом программ с уровнем ограничений, накладываемых на язык спецификаций. Для работы с неограниченной логикой первого порядка неизбежно потребуется довольно богатый запас различных правил вывода, наличие которых может привести в ходе каждого конкретного синтеза к появлению большого числа аморфных на вид и не ведущих к цели выводов.

Ряд исследователей, например, Мюррей (1978), наоборот, ограничивали язык спецификаций некоторым подмножеством языка логики первого порядка (отличным от логики дизъюнктов), за счет чего достигалось упрощение системы вывода. Можно пойти еще дальше и использовать для представления спецификаций исключительно логику хорновских дизъюнктов (как это было в нашем примере из разд. VI. 4), полагаясь тем самым только лишь на резолютивный вывод. В результате было бы достигнуто приятное единство языка программирования и языка спецификаций, а также метода исполнения программ и метода их синтеза. Существующие логические интерпретаторы можно было бы без труда приспособить к диалоговым системам синтеза программ, подобных тем, которые уже разработаны Дарлингтоном для функциональных языков. Потенциальное возражение против этого подхода заключается в том, что язык хорновских дизъюнктов представляется слишком узким, чтобы служить в качестве общего языка спецификаций. Возможно, что это так, а возможно и нет. В прошлом подобное утверждение часто приводилось в поддержку употребления неограниченного языка логики первого порядка, однако, может быть, его следует пересмотреть.

Интересной темой для исследований является возможность вывода программ, предназначенных для исполнения в параллельном режиме. Некоторые подходы к преобразованию последовательных программ в параллельные рассматривались в статье Хоггера (1982b). Может быть, стоит исследовать также возможности использования логического программирования для верификации и преобразования традиционного программного обеспечения. В этом направлении работы почти не ведутся, хотя трансляция логических программ в программы, написанные на Паскале, исследовалась Эллоком (1981).

VII. Реализация

Логические программы были впервые исполнены на компьютере в 1972 г. С тех пор было затрачено много усилий на разработку реализаций логики как языка программирования в поисках все большей их эффективности и практичности. В самом деле, темпы, с которыми развиваются сейчас новые системы с тем, чтобы сразу же обслужить возникающие потребности постоянно растущего сообщества логических программистов, а также противостоять еще не ясным, но принимающим угрожающие размеры требованиям следующего поколения архитектур ЭВМ, сводят на нет всякую попытку предложить какие-либо устойчивые принципы методологии реализации.

Поэтому здесь мы не станем делать такой попытки. Данная глава предназначена только для того, чтобы дать ориентиры начинающему разработчику, реализаций, которому в противном случае — если, конечно, он не работает вместе с опытным руководителем — пришлось бы самому постигать значительный объем основных фактов или прибегнуть к помощи научной литературы: так или иначе, его ожидала бы тяжелая задача. Тем не менее в излагаемом здесь материале многие аспекты остаются определенными не полностью, и поэтому предполагается, что читатель окажется достаточно компетентным в общих вопросах программирования и сможет сам восполнить имеющиеся пробелы.

Существует широко распространенная точка зрения, согласно которой сама логика является наилучшим языком для описания и реализации логических интерпретаторов. Вполне возможно, что это и справедливо по отношению к тем, кто уже в совершенстве владеет логикой и обладает опытом реализации языков, поскольку они легко могут представлять себе способы осуществления на конкретном уровне разнообразных абстракций, используемых в логических описаниях высокого уровня. В то же время непосвященному подобные абстракции могут показаться слишком неопределенными и поэтому они не сумеют привести к настоящему пониманию. Таким образом, ввиду того что целью данной главы является только введение в рассматриваемую область, материал в ней сознательно приводится настолько ко-

крайний, насколько это позволяет отведенное для него место. Следует подчеркнуть, что назначение столь конкретного изложения заключается в том, чтобы помочь читателю мысленно построить лишь одну работающую модель; впоследствии он сможет переформулировать отдельные практические детали или даже целую конструкцию в соответствии со своими интересами.

VII. 1. Представление состояния управления

При исполнении программ в режиме *интерпретации* все события инициируются и управляются *интерпретатором*, который работает в памяти с быстрой выборкой. Эта программа — интерпретатор — осуществляет доступ главным образом к двум важным областям данных, расположенных в памяти машины. В первой содержится некоторым подходящим способом упакованная и закодированная версия входной логической программы, и эта область остается неизменной (или «статической») в течение всего исполнения. Вторая область является чрезвычайно динамичной; она используется интерпретатором для ведения протокола своих собственных действий. Эти области обычно называют *входным массивом данных* и *стеком исполнения* соответственно. В стеке эффективным образом представлены как *состояние управления* исполнением программы (его продвижение в пространстве вычислений), так и соответствующее *состояние данных* (данные, которые на текущий момент присвоены переменным программы). В этом разделе мы будем заниматься состоянием управления, и поэтому мы начнем его с рассмотрения событий, происходящих при стандартном исполнении логической программы.

VII.1.1. Механизм исполнения

Исполнение логической программы удобно описывать на основе введенных во второй главе механизмов входа в процедуру и выхода из процедуры. Мы сделаем это описание более единообразным, трактуя исходное целевое утверждение программы

$$G : ? R_1, \dots, R_k$$

просто как еще одну процедуру: исполнение программы начинается с входа в G и заканчивается выходом из него после вычисления всех возможных решений.

В процессе исполнения входной программы из нее последовательно выбираются вызовы (начиная с первого вызова R_1 из целевого утверждения G) с целью их решения. Когда интерпре-

татор переключает свое внимание с одного вызова на другой, он вычерчивает в программе путь, который называют *траекторией управления*. Всякое удлинение траектории управления происходит либо в результате входа в процедуру, либо в результате выхода из процедуры.

Вход в процедуру происходит в процессе выполнения шага вызова процедуры. В начале каждого такого шага траектория управления будет заканчиваться в каком-то вызове P_i из некоторой процедуры (или, быть может, из исходного целевого утверждения)

$P : A \text{ если } P_1, \dots, P_i, \dots, P_m$

Если предположить, что всегда применяется стандартная стратегия исполнения, то положение дел в этот момент таково: все вызовы из процедуры P , предшествующие P_i , уже были решены (хотя не обязательно всеми возможными способами) и теперь с целью решения активируется вызов P_i .

В общем случае на вызов P_i могут потенциально отвечать несколько процедур, которые называются *кандидатами для P_i* . И теперь разработчику реализации требуется решить, как именно охарактеризовать кандидата. Наиболее простой способ, который приводит также к самой простой, но зачастую и наименее эффективной реализации, — это рассматривать в качестве кандидата для P_i всякую процедуру, чье имя соответствует имени вызова P_i ; другими словами, не проводить никакого распознавания на основе согласования параметров. Однако, как обсуждалось в гл. IV, обычно гораздо более эффективно для хранения и выбора процедур использовать какую-либо схему индексирования, базирующуюся на классификации параметров. Эта схема оказывается более избирательной в отношении определения кандидатов для вызова P_i . Тем не менее важно осознавать, что всякая схема распознавания, которая не соответствует экстремальной стратегии, заключающейся в попытке полностью согласовать (унифицировать) P_i с заголовками всех имеющихся процедур, будет обычно давать некоторое количество кандидатов, в действительности не отвечающих на вызов P_i , когда дойдет дело до их испытания. Разработчику реализации следует сбалансировать преимущества, достигаемые за счет предварительного сокращения числа унификаций, которые в будущем окажутся неудачными, с непосредственными затратами на применение схем распознавания с целью минимизации множества кандидатов.

В рассматриваемом примере может оказаться так, что некоторые из кандидатов для P_i уже были подвергнуты испытанию в предыдущих попытках решить этот вызов. Те же из них, которые остаются на текущий момент неиспробованными, можно

расположить в порядке уменьшения приоритета их выбора и обозначить посредством Q , Q' , Q'' и т. д. Таким образом, следующей будет подвергнута испытанию процедура Q , и мы допустим, что она действительно отвечает на вызов P_i . Эта процедура будет в общем случае иметь вид

$$Q : \text{В если } Q_1, \dots, Q_n$$

причем P_i унифицируется с В . Пусть теперь процедура Q вызывается с помощью P_i . В результате этого шага вызова процедуры траектория управления будет продолжена от P_i до первого вызова Q_1 из тела Q и тем самым будет осуществлен вход в процедуру Q . В то же самое время интерпретатором регистрируются определенные факты относительно данного шага, которые будут объясняться позднее. После всех этих действий интерпретатор может перейти к выполнению следующего шага вызова процедуры.

С обращения вызова P_i к процедуре Q начинается попытка решить этот вызов, которая со временем может закончиться либо успехом, либо неудачей. Она приводит к успеху, если в ходе исполнения программы будут в конце концов решены все вызовы из Q . Если это происходит, то интерпретатор осуществляет *успешный выход* из процедуры Q и передает управление назад в процедуру P к вызову P_{i+1} , который будет тогда следующим активируемым вызовом.

С другой стороны, попытка решить вызов P_i заканчивается неудачей, если оказывается, что некоторый вызов из Q решить невозможно. Если это происходит, то интерпретатор осуществляет *неудачный выход* из процедуры Q и передает управление назад в процедуру P к вызову P_i . Этот вызов активируется тогда еще раз, и, для того чтобы приступить к новой попытке его решить, испытанию подвергается следующий неиспробованный еще кандидат Q' . Такое поведение называется *возвратом после неудачи* (т. е. после неудачной попытки решить P_i с помощью процедуры Q); в терминах дерева поиска оно соответствует возврату интерпретатора из тупиковой вершины ■. Если в результате этой операции возврата окажется, что ни Q' , ни какой-либо другой из оставшихся неиспробованных кандидатов Q'' и т. д. не дает никаких решений вызова P_i , то тогда потребуется осуществлять дополнительный возврат с тем, чтобы попытаться найти новые пути решения вызова P_{i-1} или его предшественников.

Если в ходе исполнения программы будут в конце концов решены все вызовы из целевого утверждения G , то интерпретатор сначала сообщит о найденном решении, а затем передаст управление назад самому последнему из активированных вызовов, у которого остались еще неиспробованные кандидаты, и

активирует его заново для того, чтобы попытаться найти новые решения. Такое поведение после успешного выхода из G называется *возвратом после успеха*; в терминах дерева поиска оно соответствует возврату интерпретатора из успешной вершины \square . Если после успешного выхода из G не осталось больше ни одного вызова с неиспользованными кандидатами, то это значит, что все вызовы были исследованы всеми возможными способами, и, стало быть, исполнение программы заканчивается.

VII.1.2. Фреймы

В процессе исполнения программы интерпретатор должен помнить целый ряд фактов относительно тех событий, которые до сих пор имели место. Во-первых, что наиболее очевидно, для того чтобы можно было выполнить конечную цель — получить значения переменных из целевого утверждения, он должен помнить, какие данные к этому моменту переменным уже были присвоены. Мы отложим обсуждение этого вопроса до следующего раздела, а тем временем сконцентрируем свое внимание на второй категории запоминаемых фактов, связанных с построением траектории управления.

Необходимость в регистрации деталей эволюции траектории управления до своего текущего состояния отчасти обусловлена тем, что интерпретатору приходится иметь дело с операцией *перехода*, выполняемой после каждого успешного выхода из процедуры; эта операция передает управление назад к вызову, расположенному непосредственно вслед за тем, который первоначально обращался к только что оставленной процедуре. Указанная необходимость возникает также по той причине, что интерпретатору приходится осуществлять поиск с возвратом, поскольку для выполнения этой операции требуется вспоминать, какие из ранее обнаруженных возможностей еще остаются открытыми для исследования.

Регистрация подобных деталей достигается обычно за счет того, что на каждом шаге вызова процедуры интерпретатор образует и хранит в памяти *фрейм*, содержащий определенные факты относительно данного шага. Можно считать, что каждый фрейм является *признаком входа* в процедуру, вызываемую на этом шаге. О процедуре, в которую управление вошло, но из которой оно еще не вышло, говорят, что она *активна*, и по этой причине фрейм иногда называют *записью активации* — в нем регистрируется шаг, на котором некоторая процедура становится активной.

Заключение о том, какие элементы следовало бы поместить в фрейм, можно сделать, рассмотрев ту полезную информацию, которую интерпретатор мог бы извлечь из фрейма для принятия

решения относительно продолжения траектории управления. С этой целью мы предположим, что фрейм F требуется образовать для шага, на котором вызов P_i из процедуры

$P : A \text{ если } P_1, \dots, P_t, P_{t+1}, \dots, P_m$

обращается к процедуре

$Q : B \text{ если } Q_1, \dots, Q_n$

Фрейм F, стало быть, будет являться признаком входа в процедуру Q. Через некоторое время в результате решения последнего вызова Q_n управление может успешно выйти из Q. Для того чтобы интерпретатор мог в этом случае определить следующий активируемый вызов P_{t+1} , мы договоримся, что фрейм F содержит *указатель перехода* (или *указатель П*) к вызову P_{t+1} ; более точно, этот указатель мог бы состоять из адреса, по которому P_{t+1} хранится в массиве данных, представляющих входную программу. Разумеется, для получения доступа к этому указателю интерпретатор должен сначала быть в состоянии выделить соответствующий фрейм F среди всех тех фреймов, которые уже были образованы в ходе исполнения программы. Это достигается за счет принятия дополнительного соглашения о том, что каждый фрейм содержит также *указатель родительского фрейма* (или *указатель РФ*), который указывает на фрейм, являющийся *родительским фреймом* по отношению к данному. Используя наш пример, мы определим это понятие следующим образом. Рассмотрим все те шаги, на которых активируются вызовы из тела процедуры Q; тогда фрейм для каждого из них будет иметь F в качестве своего родительского фрейма, поскольку F является признаком входа в Q.

Для того чтобы увидеть, как вход в процедуру управляется с помощью этих указателей, мы рассмотрим рис. VII. I. Первые две ячейки каждого из фреймов, представленных на этом рисунке, содержат соответственно указатель РФ и указатель П. В остальных ячейках, изображенных пустыми, на самом деле могут содержаться дополнительные указатели, предназначенные для управления процессом возврата; какие именно, мы объясним позднее.

Представим себе, что интерпретатор на основании обращения к некоторому факту T в ответ на вызов Q_n только что образовал фрейм F'. Так как Q_n — последний вызов в Q, указатель П в F' будет в этом случае указывать на воображаемую позицию, находящуюся позади Q_n и обозначающую успешный выход из процедуры Q. Указатель РФ в F' будет указывать на фрейм F. В силу того, что факт T не содержит вызовов, следующим действием интерпретатора должен быть успешный выход из T. Принимая во внимание указатель П в F', он устанавливает, что

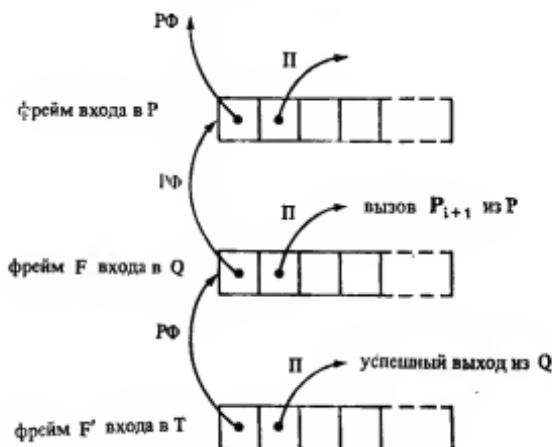


Рис. VII. 1. Информация, требуемая для успешного выхода из процедуры.

управление следует вернуть на некоторую позицию в процедуре Q. Поскольку, однако, оказывается, что эта позиция означает успешный выход из Q, необходимо выполнить еще одну операцию перехода, а именно ту, которая вернет управление к процедуре, вызывавшей процедуру Q. Указатель РФ в фрейме F' определяет более ранний фрейм F, при котором был осуществлен вход в процедуру Q, а указатель Π в F в свою очередь определяет P_{i+1} как вызов, к которому следует передать управление; нормальное исполнение программы возобновляется тогда путем активации этого вызова.

Заметим, что на рис. VII. 1 представлены три шага (входы в процедуры P, Q и T), образующие часть следующего сегмента вычисления:

$$\begin{array}{l} \vdots \\ \text{вход в } P : ? P_1, \dots, P_m, \dots \\ \vdots \\ \text{вход в } Q : ? P_1, P_{i+1}, \dots, P_m, \dots \\ \quad ? Q_1, \dots, Q_n, P_{i+1}, \dots, P_m, \dots \\ \vdots \\ \quad Q_n, P_{i+1}, \dots, P_m, \dots \\ \text{вход в } T : \\ \text{выход из } T : \\ \text{выход из } Q : ? P_{i+1}, \dots, P_m, \dots \end{array}$$

Таким образом, имеется взаимно однозначное соответствие между целевыми утверждениями в стандартном представлении вычисления в виде их последовательности и фреймами, образующими представление вычисления в интерпретаторе.

Из предыдущих обсуждений должно быть ясно, что образование, хранение и использование указателей перехода и указателей родительских фреймов дает адекватный базис для организации тех продолжений траектории управления, которые вызываются входами в процедуры и успешными выходами из них. Аналогичные механизмы используются для управления исполнением многих программ, написанных на традиционных процедурных языках. Однако дополнительное свойство недетерминированности логических программ приводит к необходимости обеспечивать, кроме того, процесс возврата. Как это делается, объясняется в следующем разделе.

VII.1.3. Механизм возврата

Рассматривая те же самые процедуры P и Q , что и прежде, предположим теперь, что после того, как был осуществлен вход в Q в ответ на вызов P_1 , некоторый вызов Q_1 оказывается неразрешимым. В этом случае интерпретатор должен быть в состоянии вспомнить тот ближайший шаг, если, конечно, он существует, на котором остались открытыми одна или несколько еще не испробованных возможностей, и он должен также суметь точно определить, какие именно эти возможности. Для того чтобы удовлетворить первому из этих двух требований, в течение всего процесса исполнения программы поддерживается едиственный регистр, называемый БТВ и всегда указывающий на образованный самым последним фрейм, соответствующий шаг которого дает по крайней мере одного неиспробованного кандидата в дополнение к тем, что уже действительно были вызваны. Такой фрейм называется точкой возврата; упомянутый регистр называется БТВ, потому что он всегда указывает на ближайшую точку возврата.

Рассмотрим момент, который непосредственно предшествует образованию фрейма F , являющегося признаком входа в процедуру Q . Регистр БТВ указывает на некоторый предыдущий фрейм F^* , оказывающийся в данный момент ближайшей точкой возврата (см. рис. VII.2a). Сразу после того как будет образован фрейм F , регистр БТВ должен быть обновлен — он должен указывать теперь на F , поскольку мы предположили, что имеются неиспробованные кандидаты Q' , Q'' и т. д., в результате чего F становится новой точкой возврата. Получаемая в итоге ситуация изображена на рис. VII.2b.

Наличие регистра БТВ и его обновление сами по себе еще не дают адекватную информацию для управления процессом возврата. Посмотрим, что произойдет, когда будет обнаружена неразрешимость вызова Q_j . Простоты ради и не теряя при этом общности, мы будем считать, что с момента образования фрейма F не возникло ни одной новой точки возврата, и, стало быть, регистр БТВ все еще указывает на F . Интерпретатор должен

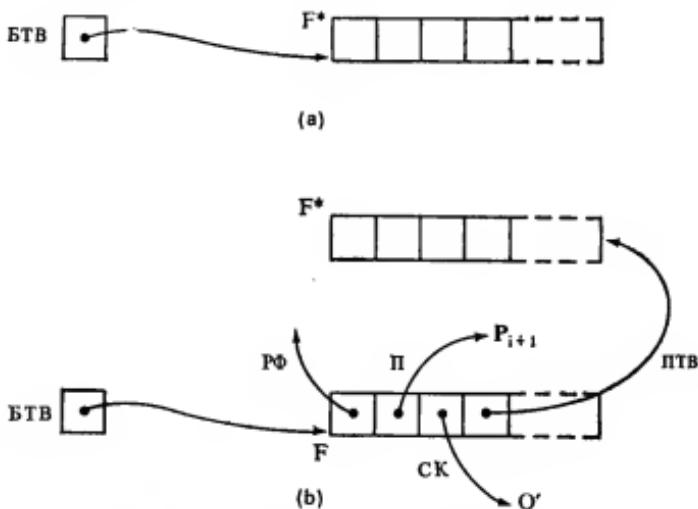


Рис. VII. 2. Образование новой точки возврата F : (а) непосредственно перед образованием F , (б) непосредственно после образования F . Сокращения расшифровываются следующим образом: БТВ — регистр ближайшей точки возврата; РФ — указатель родительского фрейма; П — указатель перехода; СК — указатель следующего кандидата; ПТВ — указатель предыдущей точки возврата; F^* — предыдущая ближайшая точка возврата.

теперь оказаться в состоянии вспомнить, что процедура Q' является кандидатом для вызова P_i , который нужно активировать следующим.

Один простой способ достижения этой цели состоит в том, чтобы в момент образования фрейма F построить *указатель следующего кандидата* (*СК*), указывающий на процедуру Q' , и хранить его в третьей ячейке F . Поэтому, когда обнаружится, что вызов Q_j неудачный, интерпретатор может справиться в регистре БТВ и найти фрейм F , а затем использовать указатель СК из F для нахождения Q' . Сам вызов P_i , который нужно заново активировать вместе с этим новым кандидатом, определяется, исходя из того факта, что указатель П из F указывает на последователя P_{i+1} вызова P_i . Как только эта информация будет извлечена из F , и сам фрейм F , и все его фреймы-потомки,

представляющие попытку решить P_1 с помощью процедуры Q , можно отбросить. На этом операция возврата после неудачи завершается, и дальнее выполнение может продолжаться обычным образом, пытаясь выполнить новый шаг вызова процедуры. Возврат после успеха происходит аналогично за исключением того, что прежде всего должно быть выдано сообщение о найденном решении целевого утверждения.

Для организаций процесса возврата требуется еще одно дополнительное средство. Рассмотрим тот момент, когда в ходе выполнения только что описанной операции возврата интерпретатор собирается отбросить фрейм F . В этот момент F является (в соответствии с нашим предположением) ближайшей точкой возврата, и регистр БТВ в настоящее время указывает именно на него. Для того чтобы после отбрасывания F интерпретатор не потерял след своих неиспользованных возможностей, регистр БТВ должен быть возвращен в предыдущее состояние, т. е. он должен указывать теперь на предыдущую точку возврата F^* . А чтобы F^* можно было с этой целью найти, мы условимся, что в момент образования фрейма F содержимое регистра БТВ (который в это время указывает на F^*) копируется в четвертую ячейку F , называемую *указателем предыдущей точки возврата* (ПТВ). Когда впоследствии осуществляется возврат, этот указатель извлекается из F и копируется обратно в БТВ; тем самым гарантируется, что знания интерпретатора о своем собственном продвижении при отбрасывании F будут должным образом сохранены.

На рис. VII.2 показана обработка информации, связанной с образованием новой точки возврата F , представляющей обращение вызова P_1 к процедуре Q . Ячейка $Р\Phi$ указывает на фрейм, образованный при входе в P ; ячейка Π указывает на вызов, который должен быть активирован после решения P_1 ; ячейка $СК$ указывает на следующего кандидата Q' , который будет испробован в ответ на вызов P_1 ; а ячейка $ПТВ$ указывает на предыдущую точку возврата F^* . Поскольку точкой возврата является F , регистр БТВ был обновлен и указывает теперь на F . Если бы фрейм F не был точкой возврата, то он не имел бы ни ячейки $СК$, ни ячейки $ПТВ$, а регистр БТВ так и продолжал бы указывать на фрейм F^* .

VII.1.4. Шаг вызова процедуры

В начале каждой попытки выполнить шаг вызова процедуры интерпретатор выберет для активации некоторый вызов. Мы условимся, что указатель на этот вызов содержится в регистре с именем $ТВ$ («текущий вызов»). Еще один регистр, называемый $СКА$ («следующий кандидат»), указывает на следующего кан-

дидата — процедуру, которая в следующий раз будет испробована в ответ на упомянутый вызов. Для того чтобы избежать многократных ссылок на указатели, мы будем свободно пользоваться этими именами регистров, как будто они прямо относятся к указываемым ими объектам; так, например, говоря «вызов ТВ», мы на самом деле имеем в виду «вызов, на который указывает регистр ТВ».

Следующая задача интерпретатора — найти процедуру, отвечающую на ТВ. Точные детали поиска зависят, разумеется, от разработчика реализации. Предположим, однако, что с этой целью используется некоторый вид схемы индексирования. Исследуя параметры ТВ, интерпретатор среди всех доступных процедур, имеющих соответствующее имя, может выделить некоторое (быть может, пустое) подмножество тех процедур, которые потенциально отвечают на ТВ. Просматривая этих кандидатов, расположенных в исходном текстуальном порядке, интерпретатор ищет затем первую процедуру, которая (а) действительно отвечает на ТВ и (б) совпадает с кандидатом СКА или расположена (в смысле текстуального упорядочения) после него.

В результате этого поиска достигаются две цели. Во-первых, еще в один регистр с именем ТП («текущая процедура») помещается указатель, который будет указывать на выбранную процедуру, отвечающую на ТВ, если таковая найдется, и будет пустым указателем \perp в противном случае. Во-вторых, если найдена отвечающая на ТВ процедура, то регистр СКА обновляется — он будет указывать теперь на кандидата, расположенного вслед за выбранной процедурой, если такой существует; в противном случае он устанавливается в состояние \perp . Таким образом, если некоторая процедура ТП отвечает на вызов ТВ, то обновление СКА гарантирует, что в случае последующего возврата интерпретатора и повторной активации ТВ в ответ на этот вызов не будет вызываться еще раз ТП — вместо нее должна быть найдена новая процедура, расположенная на позиции нового кандидата СКА или после него.

Мы можем, таким образом, различать три следующих возможных результата текущей активации ТВ:

(i) ТП = \perp . Ни процедура СКА, ни процедуры, расположенные после нее, не отвечают на ТВ, и, стало быть, непосредственный шаг вызова процедуры оказывается невозможным; для поиска более раннего подходящего для активации вызова (т. е. вызова, у которого имеются ненпробованные кандидаты) требуется выполнять процесс возврата; если этот процесс будет успешным, то ТВ обновляется путем замены его на вновь найденный вызов и предпринимается новая попытка вызова процедуры; в противном случае исполнение программы заканчивается.

(ii) $T\Phi \neq -l$, а $CKA = -l$. Имеет место детерминированный шаг вызова процедуры, на котором ТВ вызывает ТП, образуя новый фрейм, являющийся признаком входа в ТП; в этом фрейме нужны только две ячейки РФ и П.

(iii) $T\Phi \neq -l$ и $CKA \neq -l$. Имеет место (потенциально) не-детерминированный шаг вызова процедуры, на котором ТВ вызывает ТП, образуя новый фрейм, указывающий на вход в ТП; этот фрейм является точкой возврата, и поэтому потребуются все его четыре управляющие ячейки.

VII.1.5. Алгоритм управления

Для элементарного интерпретатора, реализующего стандартную стратегию, алгоритм управления оказывается очень простым. Он состоит из механизмов, необходимых для осуществления выбора вызова, выбора процедуры и процесса возврата. Организация этих действий зависит от соответствующего образования и использования фреймов, представляющих состояние исполнения.

Фреймы удобно хранить в хронологическом порядке в *стеке*, расположенному в памяти машины с быстрой выборкой. В результате каждого входа в процедуру к стеку добавляется новый фрейм, тогда как в процессе возврата из него удаляется один или несколько фреймов. (Другие возможности для удаления фреймов будут обсуждаться в данной главе позднее.)

Помимо информации, которая хранится в стеке, полезно использовать также несколько отдельных регистров, содержащих важные детали относительно состояния исполнения программы. Некоторые из них уже были определены. Непосредственно после образования нового фрейма в этих регистрах содержится следующая информация. Регистр *БТВ* указывает на ближайшую точку возврата, если она имеется; в противном случае он устанавливается в состояние $-l$. Регистр *ТП* указывает на процедуру, в которую осуществлен вход при образовании этого фрейма; ТВ указывает на вызов, который обращался к упомянутой процедуре. Регистр *СКА* указывает на следующего неиспользованного кандидата для ТВ, если он существует; в противном случае регистр устанавливается в состояние $-l$. Новый регистр с именем *БП* («ближайший предок») указывает на родительский фрейм рассматриваемого фрейма, а еще один новый регистр *f* используется для хранения номера позиции фрейма в стеке — таким образом, индексированное имя вида *РФ* (*f*) обозначает ячейку *РФ* в этом фрейме.

Полный алгоритм управления представлен на рис. VII.3 с помощью традиционной алголоподобной нотации. Блоки операторов отделяются друг от друга обычными абзацами, а не опе-

Шаг 1 (Установка в исходное состояние)

```
f := 1
TP := → вводное целевое утверждение
BP := ⊥
BTB := ⊥
RF(1) := ⊥
Π(1) := выход
```

Шаг 2 (Выбор вызова)

```
If TP → есть факт
then TB := Π(f)
    while TB = выход and BP ≠ 1
        do TB := Π(BP)
            BP := RF(BP)
    If TB = выход
        then вывод решения
            целевого утверждения
            goto Шаг 5
    else TB := → первый вызов из TP →
        BP := f
    If для TB → не имеется кандидатов
    then goto Шаг 5
    else СКА := → первый кандидат
```

Шаг 3 (Выбор процедуры)

Используя текущие состояния TB и СКА, найти отвечающую на TB процедуру, а также других не испробованных до сих пор кандидатов (как это описано в разд. VII.4) и таким образом обновить TP и (возможно) СКА. Затем

```
If TP = ⊥
then goto Шаг 5
```

Шаг 4 (Образование фрейма)

```
f := f + 1
RF(f) := BP
If TB → есть последний вызов
    в процедуре
then Π(f) := выход
else Π(f) := → вызов, следующий
    за TB →
If СКА ≠ ⊥
then СК(f) := СКА
    ПТВ(f) = BTB
    BTB := f
goto Шаг 2
```

Комментарии

Образуется первый фрейм для входа в целевое утверждение (которое, как предполагается, имеет по крайней мере один вызов). Этот фрейм точкой возврата не является.

Комментарии

Если TP является фактом, то следующий вызов нужно искать, применяя одну или несколько операций перехода — если они результата не дадут, то решение целевого утверждения найдено. В противном случае в качестве TB выбирается первый вызов из TP. Поскольку текущий фрейм f является признаком входа в TP, после выполнения данного шага он станет ближайшим предком, и потому регистр BP должен быть обновлен на f. Если TB не имеет кандидатов, то следует выполнять процесс возврата; в противном случае поместить в СКА первый из них.

Комментарии

Предпринимается попытка унифицировать TB с заголовками его кандидатов, начиная с СКА. Неудачный исход ее приводит к процессу возврата, а при успехе TP определяется как первый отвечающий на TB кандидат, а СКА — как следующий за ним неиспользованный кандидат, если, конечно, он существует.

Комментарии

Образуется фрейм для шага, на котором TB вызывает (входит в) TP. Фрейм помещается в стеке на позиции с номером f + 1, его ячейки RF и Π строятся при помощи BP и TB. Если СКА ≠ ⊥, то для TB имеются неиспользованные еще кандидаты, и потому только что образованный фрейм является точкой возврата; в этом случае ячейки СК ПТВ строятся при помощи СКА и BTB, а регистр BTB затем обновляется — теперь он будет указывать на этот новый фрейм.

<u>Шаг 5 (Возврат)</u>	<u>Комментарии</u>
<pre> If БТВ = - then закончить исполнение программы else СКА := СК(БТВ) ТВ := → вызов, предшес- твующий П(БТВ) БП := РФ(БТВ) f := БТВ - 1 БТВ := ПТВ(БТВ) goto Шаг 3 </pre>	<p>Если БТВ = - , то возврат невозможен и исполнение программы заканчивается. В противном случае для нахождения активируемого еще раз вызова и следующего кандидата для него используется ближайшая точка возврата. В результате уменьшения значения f до $\text{БТВ} - 1$ эффективным образом отбрасывается фрейм, соответствующий этой точке возврата, а также все фреймы, следующие за ним.</p>

Рис. VII. 3. Алгоритм управления.

раторными скобками **begin ... end**. Символ «выход» обозначает позицию успешного выхода, расположенную в конце каждой процедуры; символ := обозначает деструктивное присваивание; запись $r \rightarrow$ обозначает объект, на который указывает некоторый регистр r , в то время как $\rightarrow x$ обозначает указатель на объект x . Заметим, что, хотя ячейки ТВ и П явно определяют вызовы, тем не менее в них неявно определяются также содержащие эти вызовы процедуры. В частности, когда им присваивается значение «выход», эта позиция должна включать в себя идентификатор процедуры, на выход из которой указывается в данных ячейках.

VII.2. Представление присваиваний данных

Вопрос о том, как наилучшим образом обращаться с присваиванием данных переменным программы, долгое время привлекал внимание разработчиков реализаций, и, быть может, на этот вопрос простого ответа не существует. Выбор какого-либо метода заключается в принятии решения относительно того, как сбалансировать соответствующие затраты на построение, хранение, выборку и отбрасывание данных, что в свою очередь связано с тем видом программ, которые, как предполагается, будут исполняться интерпретатором. В этом разделе наиболее важные соображения описываются с помощью ряда простых конкретных примеров исполнения логических программ.

VII.2.1. Одностековое представление

В качестве первого примера ниже приводится программа 39, представляющая собой упрощенный вариант программы 13 (гл. III) и имеющая целью нахождение всех таких строк вида

$u.A.w$, которые являются префиксами строки $B.A.A.C.NIL$. Префикс каждой строки S — это либо пустая строка NIL , либо произвольная подстрока S , начинающаяся с ее первого элемента.

Программа 39

G1 : ? префикс($u.A.w, B.A.A.C.NIL$)

P1 : префикс(NIL, y)

P2 : префикс($v.x, v.y$) если префикс(x, y)

На рис. VII.4 изображено дерево поиска, исследуемое путем применения стандартной стратегии; оно содержит три различных

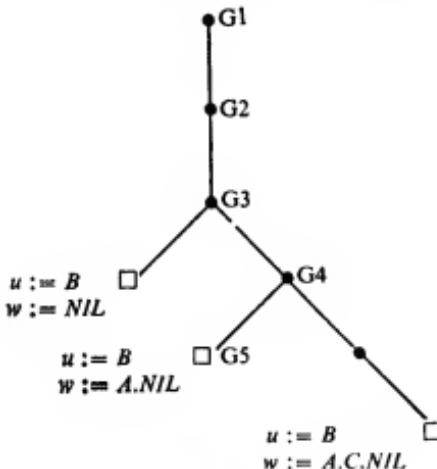


Рис. VII.4. Вычисления по программе 39.

решения нашей задачи. В частности, вычисление $G1 - G5$, которое дает второе решение, выбрано для детального анализа содержащихся в нем присваиваний данных. Состояния последовательных целевых утверждений и присваивания целевым переменным в вычислении $G1 - G5$ таковы:

G1 : ? префикс($u.A.w, B.A.A.C.NIL$)

{ $u := B$ }

G2 : ? префикс($A.w, A.A.C.NIL$)

G3 : ? префикс($w, A.C.NIL$)

{ $w := A.x$ }

G4 : ? префикс($x, C.NIL$)

{ $x := NIL$ }

G5 : \square ответ $u := B, w := A.NIL$

На каждом из первых трех шагов вызывается процедура P2, в которую входят переменные v , x и y . Так как всякий вход

в Р2 приводит к новому использованию этих переменных, то отсюда вытекает, что каждый вход должен повлечь за собой и дополнительное распределение памяти, необходимой для хранения каких бы то ни было присваиваемых им данных. Более точно, при каждом входе в процедуру для ее переменных, если, конечно, они имеются, должна выделяться новая память. (На практике тем не менее бывают ситуации, когда для переменной, встречающейся в процедуре в некотором особом контексте, не требуется выделять какой-либо долговременной памяти, поскольку ее значение, используемое лишь временно в ходе выполнения шага унификации при вызове этой процедуры, никогда впоследствии не упоминается.) В силу того, что каждое такое распределение памяти сопровождается образованием нового фрейма, обычно делают так, чтобы значения переменных помещались в самом этом фрейме — по одному в ячейке. Таким образом, в общем случае каждый фрейм наряду с ячейками управления будет содержать еще ячейки переменных. Подобная организация дает одностековое представление исполнения, которое образует адекватный базис для реального, хотя и довольно незамысловатого, интерпретатора.

Основное содержимое представления вычисления G1 — G5 в виде единственного стека показано в приводимой ниже таблице, где запись вида $\uparrow(P2)$ обозначает позицию успешного выхода из указанной процедуры P2.

Вызывающая процедура	Позиция фрейма	Содержимое фрейма				
		РФ	П	СК	ПТВ	Переменные
G1	1	—	—	—	—	$w_1 := B, w_1 := A.x_4$
P2	2	1	$\uparrow(G1)$	—	—	$v_2 := B, x_2 := A.w_1,$ $y_2 := A.A.C.NIL$
P2	3	2	$\uparrow(P2)$	—	—	$v_3 := A, x_3 := w_1,$ $y_3 := A.C.NIL$
P2	4	3	$\uparrow(P2)$	—	—	$v_4 := A, x_4 := NIL,$ $y_4 := C.NIL$
P1	5	4	$\uparrow(P2)$	$\rightarrow P2$	—	$y_5 = C.NIL$

Непосредственно после образования фрейма 5 регистры находятся в следующих состояниях

БП = 4 (фрейм 4 является ближайшим родительским фреймом)

БТВ = 5 (фрейм 5 является ближайшей точкой возврата)
f = (фрейм 5 был образован последним)

ТП = указатель на Р1 (процедуру, в которую только что осуществлен вход)

ТВ = указатель на первый вызов из Р2 (который обращается к ТП)

Каждый фрейм в стеке изображается одной строкой в таблице. Та часть фрейма, которая предназначена для переменных, на самом деле состоит из набора ячеек — по одной для каждой из переменных, имеющих вхождения в вызванную процедуру. На любом этапе эволюции стека содержимое ячейки, предназначеннной для переменной, может быть просто некоторым специальным символом, скажем *, означающим, что этой переменной на данный момент еще не присвоено никакого значения. С другой стороны, переменной может быть присвоено некоторое значение данных; если это значение достаточно простое (например, константа небольшой длины), то его можно хранить прямо в ячейке рассматриваемой переменной; в противном случае его следует хранить в каком-то другом месте памяти, а в ячейку переменной помещать тогда просто соответствующий указатель. Отметим также, что значение, хранимое в ячейке переменной, может быть указателем на некоторую другую ячейку переменной; например, указатель на ячейку для w_1 хранится в ячейке переменной x_3 .

Хотя имена переменных и приведены в таблице для удобства читателя, тем не менее в явном виде они в фрейме не содержатся. Вместо этого каждый идентификатор переменной устанавливается неявно по номеру позиции ее ячейки в фрейме. Так, например, интерпретатор может считать, что переменным v , x и y из процедуры Р2 в каждом фрейме, указывающем на вход в Р2, всегда сопоставлены первая, вторая и третья ячейки переменных соответственно. Заметим, что, выделяя новое множество ячеек переменных при каждом входе в процедуру, мы никогда не перепутаем, скажем, значения, которые получает переменная x при двух разных входах в процедуру Р2. В каждом случае в идентификатор переменной включается идентификатор соответствующего фрейма, как это показывают имена x_2 , x_3 и т. д. в приведенной выше таблице. Подобная классификация автоматически удовлетворяет исходным требованиям переименования переменных, введенным в гл. I во избежание путаницы их идентификаторов.

Сразу после образования фрейма его ячейки переменных будут содержать различные значения или находиться в состоянии * в зависимости от результатов выполнявшейся на этом шаге унификации. На самом деле в момент распределения ячеек для переменных u_1 , w_1 и x_4 этим переменным не присвоено еще никаких значений, однако позднее им присваиваются соответственно значения B (при образовании фрейма 2), $A.x_4$ (при об-

разовании фрейма 4) и NIL (при образовании фрейма 5). Отсюда видно, что в результате образования фрейма данные могут присваиваться ячейкам, расположенным в каких-то предыдущих фреймах: присваивания этого вида называются *выходными присваиваниями*, поскольку каждое из них передает выходные данные из вызываемой процедуры обращавшемуся к ней вызову.

Вспоминая поведение алгоритма управления, заметим, что процедура ($P1$), в которую осуществлялся вход при образовании фрейма 5, является фактом. Это обстоятельство вызывает последовательность выходов из процедур, которая ведет назад к выходу из $G1$, означающему, что целевое утверждение решено. Решение целевого утверждения восстанавливается из содержимого ячеек переменных в фрейме 1, а также всех других ячеек, на которые в первых имеются ссылки. Так, например, для того чтобы построить и выдать решение $u_1 := B$, $w_1 := A.NIL$, потребуется выяснить содержимое ячеек переменных u_1 , w_1 и x_4 .

После этого интерпретатор должен выполнять возврат. Из предыдущих рассмотрений данного процесса мы уже знаем, что все фреймы, начиная с ближайшей точки возврата (в нашем примере к ним относится только фрейм 5, так как оказывается, что регистр БТВ указывает на него), будут тогда отброшены. Теперь возникает новая проблема. Отбрасывание фрейма означает, что забывается тот вклад в исполнение программы, который был сделан на соответствующем ему шаге. Что же следует нам предпринять для того, чтобы забыть все выходные присваивания, полученные в момент образования этого фрейма? Более конкретно, в результате образования фрейма 5 получилось присваивание $x_4 := NIL$; когда фрейм 5 отбрасывается, это присваивание должно быть уничтожено посредством возвращения переменной x_4 в состояние *. В следующем разделе мы опишем, каким образом интерпретатор запоминает, что он должен выполнить эти действия.

VII.2.2. След

Операция возврата удаляет из стека как ближайшую точку возврата (на которую указывает регистр БТВ), так и фреймы, расположенные вслед за ней (т. е. образованные после нее). В соответствии с этой операцией должны быть уничтожены те присваивания (если, конечно, они имеются), которые в ходе образования удаленного теперь фрейма сопоставляли данные ячейкам переменных в фреймах, предшествующих точке возврата. Для того чтобы интерпретатор помнил об этих присваиваниях, в период исполнения образуется еще одна структура, называемая *следом* (или «списком восстановления»). В ней регистри-

руются идентификаторы переменных, которым были сделаны такие присваивания. В процессе исполнения программы поддерживается регистр ВС («вершина следа»), всегда указывающий на вершину следа, т. е. на его самый последний элемент. В начале исполнения след пуст, и регистр ВС содержит пустой указатель \perp .

Точные детали добавления к следу новых элементов таковы. Предположим, что образуется некоторый фрейм. Во-первых, если этот фрейм является точкой возврата, то *указатель следа* (СЛ), установленный в состояние ВС, хранится в фрейме в пятой ячейке управления. Во-вторых, независимо от того, является фрейм точкой возврата или нет, если его образование приводит (вследствие выполнявшейся на этом шаге унификации) к присваиванию данных переменной, ячейка которой расположена в

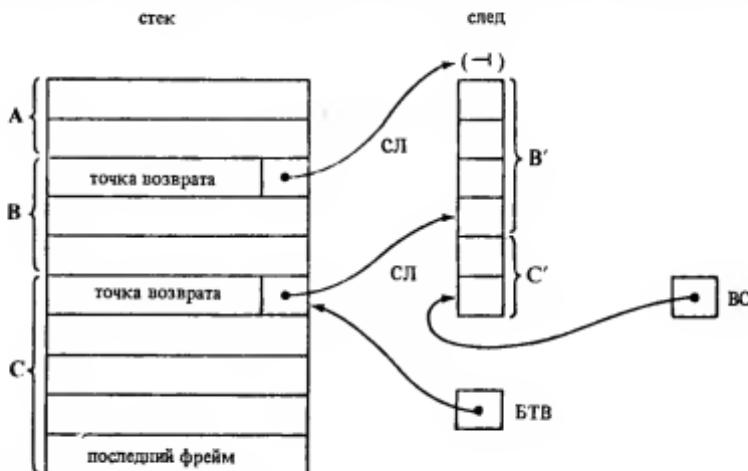


Рис. VII. 5. След.

фрейме, предшествующем ближайшей на текущий момент точке возврата, то идентификатор этой переменной (а точнее, указатель на сопоставленную ей ячейку) помещается в след, и соответствующим образом изменяется регистр ВС.

В общих чертах этот механизм иллюстрируется на рис. VII. 5, где показаны две точки возврата в текущем стеке, причем каждая из них содержит указатель СЛ, определяющий некоторую позицию в следе.

Если теперь (т. е. непосредственно вслед за состоянием, изображенным на рис. VII. 5) будет выполняться операция возврата, то все ячейки переменных, идентификаторы которых расположены в следе после (но не на) позиции, определяемой

СЛ(БТВ), должны быть возвращены в состояние *. На рисунке эти идентификаторы содержатся в сегменте следа С'. Сами ячейки входят в различные фреймы, расположенные в сегментах стека А и В (но не в С), и ранее им были присвоены данные посредством образования фреймов из сегмента С. В результате выполнения операции восстановления регистр ВС возвращается в состояние СЛ(БТВ) (и при этом эффективным образом отбрасывается сегмент следа С'), затем регистр БТВ возвращается в состояние ПТВ(БТВ), а сегмент С из стека удаляется. На этом операция возврата заканчивается, и исполнение программы возобновляется обычным образом. Для читателя не должно составить труда представить себе, как эти мероприятия могут быть включены в алгоритм управления, приведенный на рис. VII.3.

Вернемся к рассмотрению нашего предыдущего примера. Состояние следа после образования фрейма 5 таково, что в нем содержится только один элемент, определяющий ячейку переменной x_4 , и регистр ВС указывает на него. Регистр БТВ указывает на фрейм 5, в котором указатель следа СЛ(БТВ) есть $\neg i$. Когда происходит возврат, в сегменте следа, определяющего ячейки переменных, которые должны быть восстановлены, содержится лишь элемент x_4 . В соответствии с этим ячейка переменной x_4 возвращается в состояние *, регистр ВС возвращается в состояние СЛ(БТВ) = $\neg i$, затем регистр БТВ возвращается в состояние ПТВ(БТВ) = $\neg i$ и, наконец, фрейм 5 отбрасывается.

VII.2.3. Представление данных

В зависимости от обстоятельств элемент данных, который присваивается переменной посредством унификации, может быть либо константой, либо переменной, либо структурированным термом. Вообще говоря, присваиваемую константу не большой длины можно хранить непосредственно в ячейке соответствующей переменной. Если же размеры константы слишком большие, то ее следует хранить в каком-либо другом месте памяти, а в ячейку переменной помещается указатель на нее. В случае присваивания одной переменной значения другой переменной, такого как $x_3 := w_1$, в ячейке для x_3 просто хранится указатель на ячейку для w_1 .

Более сложным делом оказывается присваивание структурированного терма. При наиболее простой схеме в ячейке переменной хранится указатель на некоторый кластер последовательных ячеек, представляющих терм. Рассмотрим в качестве примера какое-либо произвольное присваивание, скажем $z := A.B.C.NIL$. На рис. VII.6а показано представление терма $A.B.C.NIL$ с помощью семи ячеек. В каждой из них содержится либо констан-

та небольшой длины, либо функтор вместе с числом его аргументов. Первая ячейка определяет самый внешний функтор, которым является «точка», и указывает (посредством числа 2), что в двух следующих ячейках представлены аргументы этого функтора; они в свою очередь определяются как константа *A* и еще один структурированный терм, который представлен аналогичным образом, начиная с третьей ячейки. Итак, в общем случае в ячейке переменной может содержаться либо символ *, означающий отсутствие какого-либо присваивания, либо константа небольшой длины, либо указатель на другую ячейку переменной, либо указатель на кластер, представляющий структурированный терм. Поскольку интерпретатор должен быть в состоянии различать все эти возможности, в каждой ячейке несколько разрядов предназначается для помещения «кода типа», который описывает тип элемента данных, закодированного в оставшихся разрядах.

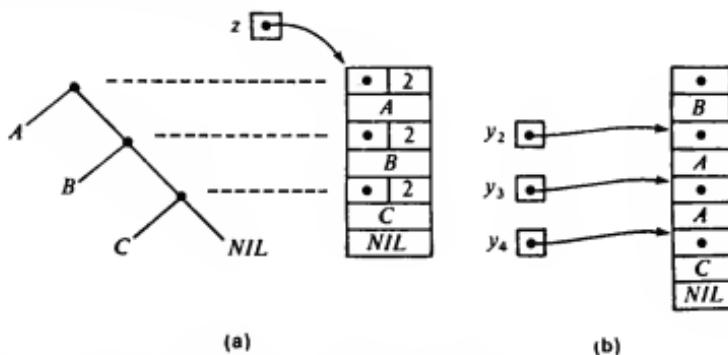


Рис. VII. 6. Представление структурированных термов:
 (а) представление $z := A.B.C.NIL$;
 (б) представление $y_2 := A.A.C.NIL$, $y_3 := A.C.NIL$, $y_4 := C.NIL$.

Рассмотрим еще раз фреймы, порождаемые программой **префикс**. Очевидно, что среди содержащихся в них присваиваемых термов существует ряд повторяющихся подструктур. А именно они встречаются в присваиваниях $y_2 := A.A.C.NIL$, $y_3 := A.C.NIL$ и $y_4 := C.NIL$. На самом деле все эти присваиваемые термы оказываются подтермами терма $B.A.A.C.NIL$, имеющего вхождение во входную программу. Отсюда вытекает один полезный способ экономии памяти, распределяемой для значений данных: как показано на рис. VII. 6b, ячейки переменных y_2 , y_3 и y_4 могут просто указывать на соответствующие части представления терма $B.A.A.C.NIL$ в массиве данных, содержащем входную программу. Кроме очевидной выгоды, связанной с экономней памяти, которую дает использование одной хранимой структуры

для представления нескольких подструктур, эта схема демонстрирует гораздо более важный общий принцип — во время исполнения логической программы для представления значений данных, присваиваемых ее переменным, не обязательно требуется какое-либо распределение памяти помимо той, которая в начале исполнения была выделена для хранения входных утверждений. В связи с этим наблюдением целесообразно обсудить теперь один важный метод чрезвычайно компактного представления структурированных данных. Упомянутый метод известен как совместное использование структур; он применяется во многих существующих реализациях.

VII.2.4. Совместное использование структур

Выгоду от совместного использования структур можно продемонстрировать, рассмотрев новую программу, которая выполняет итеративное обращение списка и получается в результате упрощения программы 4 из гл. III.

Программа 40

```
G1 : ? обратить*(NIL, A.B.C.NIL, y1)
P1 : ? обратить*(z, NIL, z)
P2 : ? обратить*(x, u.w, y) если обратить*(u.x, w, y)
```

Здесь предикат **обратить**^{*}(v_1, v_2, v_3) означает, что список v_3 является обратным по отношению к списку, получаемому добавлением v_2 в конец обращенного списка v_1 . Тем самым целевое утверждение G1 эффективным образом требует найти список y_1 , обратный по отношению к списку *A.B.C.D.NIL*.

В силу того что эта программа детерминированная, она дает в точности одно вычисление.

```
G1 : ? обратить*(      NIL, A.B.C.D.NIL, y1) (посредством
                                         входа в G1)
G2 : ? обратить*(      A.NIL,   B.C.D.NIL, y1) (посредством
                                         входа в P2)
G3 : ? обратить*(      B.A.NIL,   C.D.NIL, y1) (посредством
                                         входа в P2)
G4 : ? обратить*(      C.B.A.NIL,       D.NIL, y1) (посредством
                                         входа в P2)
G5 : ? обратить*(D.C.B.A.NIL,           NIL, y1) (посредством
                                         входа в P2)
G6 : □     ответ y1 := D.C.B.A.NIL (посредством
                                         входа в P1)
```

Это вычисление порождает шесть фреймов — по одному для каждого входа в процедуру. Записанные в них присваивания выглядят следующим образом.

Фрейм	Присваивания
1	$y_1 := *$ в начале, но $D.C.B.A.NIL$ в конце
2	$x_2 := NIL, u_2 := A, w_2 := B.C.D.NIL, y_2 := y_1$
3	$x_3 := A.NIL, u_3 := B, w_3 := C.D.NIL, y_3 := y_1$
4	$x_4 := B.A.NIL, u_4 := C, w_4 := D.NIL, y_4 := y_1$
5	$x_5 := C.B.A.NIL, u_5 := D, w_5 := NIL, y_5 := y_1$
6	$y_1 := D.C.B.A.NIL$ (выходное присваивание), $z_6 := y_1$

Здесь, очевидно, снова среди данных, присваиваемых переменным y_i , x_i и w_i , имеется много повторяющихся подструктур. Данные для переменных w_i можно представить компактно, помещая в ячейки для w_i указатели на соответствующие компоненты представления входного терма $A.B.C.D.NIL$ в виде кластера аналогично тому, как это показано на рис. VII.6б для программы префикс. Тот же метод, однако, уже нельзя применить к данным для переменных y_1 и x_i , поскольку терм $D.C.B.A.NIL$ во входной программе не встречается. При поверхностном взгляде отсюда вытекает, что для размещения этих данных должна быть выделена дополнительная память. Тем не менее оказывается, что это не так: мы увидим, что метод совместного использования структур обладает замечательным качеством — структурированные данные представляются лишь при помощи массива данных, содержащего входную программу, и ячеек переменных в стеке.

Для того чтобы увидеть, каким путем это достигается, рассмотрим, как в ходе образования фрейма 3 было порождено значение $A.NIL$, присвоенное переменной x_3 . На соответствующем шаге вызова процедуры в ответ на вызов **обратить***($A.NIL, B.C.D.NIL, y_1$) произошло обращение к процедуре Р2. Откуда возник этот вызов? В результате предыдущего входа в Р2 был образован фрейм 2 и выполнены присваивания $x_2 := NIL, u_2 := A, w_2 := B.C.D.NIL$ и $y_2 := y_1$; поэтому следующим активируемым вызовом является вызов **обратить***(u, x, w, y) из процедуры Р2 с учетом указанного выше распределения значений данных по его переменным, что дает **обратить***($A.NIL, B.C.D.NIL, y_1$). Участвующая в образовании фрейма 3 унификация присвоит тогда переменной x_3 значение фактического параметра $A.NIL$ из этого вызова, который, как мы только

что видели, получается в результате распределения в терме $i.x$ из входной программы значений данных, хранимых в ячейках переменных i и x в фрейме 2. Стало быть, все, что требуется для представления присваивания $x_3 := A.NIL$ — это указатель в ячейке переменной x_3 на терм $i.x$ из массива данных, представляющих входную программу, и второй указатель также в ячейке для x_3 на фрейм 2.

Первый указатель называется *указателем скелета*, поскольку он указывает на скелет $i.x$, который, вообще говоря, может быть любым структурированным термом, встречающимся во входной программе. Второй указатель называется *указателем среды*, поскольку он указывает на *среду* (т. е. на контекст), в которой был использован скелет. Когда при входе в процедуру образуется фрейм, то тем самым создается новая среда, состоящая из значений переменных этой процедуры. Первоначальные значения упомянутых переменных определяются унификацией, выполнявшейся при входе в процедуру, однако впоследствии они могут быть модифицированы посредством выходных присваиваний, полученных в результате решения вызовов из тела процедуры. Когда активируется один из указанных вызовов, его фактические параметры являются в этот момент результатами подстановки вместо всех тех переменных, которые присутствуют во входной форме (или в «чистом коде») вызова, их значений, определяемых текущей средой вызова, т. е. как раз состоянием ячеек переменных, хранящихся в образованном при входе в процедуру фрейме. Таким образом, когда после входа в процедуру Р2 и образования фрейма 2 активируется вызов *обратить** ($i.x, w, y$), его средой будет $\{x_2 := NIL, u_2 := A, w_2 := B.C.D.NIL, y_2 := y_1\}$ и, следовательно, его первым фактическим параметром оказывается терм $A.NIL$. Для того чтобы присвоить это значение переменной x_3 , нужно просто указать как на скелет $i.x$, так и на среду фрейма 2. Пара такого рода указателей называется *молекулой*, и ее можно рассматривать как элемент данных, конкретное значение которого устанавливается при помощи этих указателей.

Более полную картину метода совместного использования структур дает рис. VII.7, на котором изображено заключительное состояние данных, присвоенных переменным после завершения вычисления по программе 40. Каждая ячейка переменной здесь содержит либо константу, либо указатель на другую ячейку переменной, либо указатель на кластер без переменных, либо молекулу. В частности, все ячейки переменных y_1 и x_1 (за исключением x_2) содержат молекулы; ради простоты их указатели среды представлены в виде порядковых номеров позиций, которые занимают в стеке фреймы, содержащие эти среды — в действительности указателями были бы адреса в стеках, непосред-

ственно определяющие местоположение фреймов. Хотя ячейки изображены на рисунке в виде последовательных групп, на самом деле они, конечно же, расположены в соответствующих им фреймах.

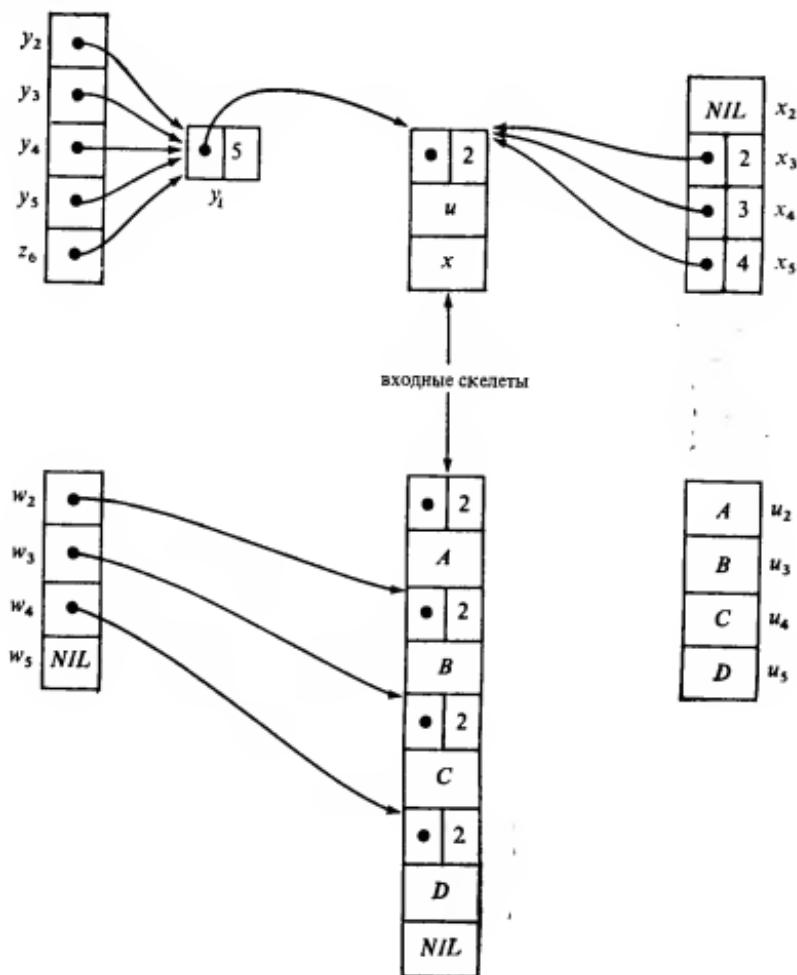


Рис. VII. 7. Представление данных, вычисленных программой обращение списка, с помощью совместного использования структур.

Конкретное значение молекулы на каждом этапе исполнения программы можно определить путем вычисления компонент, на которые ссылаются ее указатели; в свою очередь эти компоненты также могут потребовать обращения к другим молекулам.

Например, заключительное значение переменной x_4 на рис. VII.7 можно получить, подставляя в скелет *и.х* значения переменных *и* и *х*, определяемых для u_3 и x_3 средой фрейма 3. Этот процесс иногда называют «наполнением» или «интерпретацией» скелета. Значение переменной u_3 есть *B*, тогда как значением переменной x_3 является другая молекула, значением которой оказывается терм *A.NIL*. Стало быть, значение переменной x_4 есть *B.A.NIL*. Вычисление данных путем замены указателей (ссылок) на их компоненты самими компонентами обычно принято называть *разыменованием*. Очевидная ситуация, в которой необходимо производить полное разыменование, встречается тогда, когда пользователю требуется выдать структурированное решение целевого утверждения. Так, например, заключительное значение, вычисляемое программой 40 для переменной y_1 , является (в представлении с помощью совместного использования структур) *делокализованным* по нескольким связанным фреймам, и, для того чтобы предоставить пользователю вразумительный ответ *D.C.B.A.NIL*, это значение требуется полностью разыменовать.

Название метода «совместное использование структур» отражает тот факт, например, что значения переменных x_3 и x_4 имеют *общий* входной скелет *и.х* — и в этом состоит основной источник значительной компактности хранения данных, получаемой за счет применения указанного метода. Очевидно, что наибольшую пользу он приносит в том случае, когда на допустимые в период исполнения объемы памяти накладываются существенные ограничения или же память слишком дорогостоящая. За получаемую экономию памяти возможно придется расплачиваться увеличением времени обработки, необходимого для выполнения унификации, поскольку интерпретатору с целью сравнения фактических и формальных параметров может потребоваться исследовать сколь угодно длинные цепочки молекул. Это наказание будет незначительным, если в ходе исполнения программы подобные унификации выполняются редко или если вычислительная машина, на которой осуществляется реализация, обеспечена высокоэффективными механизмами косвенной адресации. Первое из этих двух условий зависит от вида исполняемой программы. При исполнении некоторых программ могут строиться большие структурированные термы, и, тем не менее, на программу унификации не ложится почти никакой нагрузки, связанный с *выборкой* данных. В других же программах могут вообще не употребляться структурированные термы, а обрабатываться, быть может, данные, представленные в виде фактов. Проблема выбора между методом совместного использования структур и какими-либо иными методами, исходя из имеющейся вычислительной машины и предполагаемой сферы применений,

беспокоят умы многих квалифицированных специалистов в области реализации и продолжает оставаться предметом многочисленных дискуссий.

VII.3. Экономия памяти

Пространство, требуемое интерпретатором для реализации стека в ходе исполнения программы, может вырасти до неприемлемых размеров, если экономическому расходованию памяти уделяется недостаточно внимания. За сокращение требуемой интерпретатору в период исполнения памяти, насколько это оказывается практически возможным, ответственность разделяют и программист и разработчик реализации, поскольку ни один из программистов не сможет писать эффективные программы для неэффективного интерпретатора, и ни один из специалистов не может разработать интерпретатор, извлекающий оптимальное поведение из всех мыслимых программ.

Само по себе совместное использование структур является экономичным методом расходования памяти для представления состояния исполнения. Тем не менее описываемый до настоящего момента элементарный интерпретатор, реализованный без каких-либо дополнительных усовершенствований, использовал бы память весьма неэффективно. Основная причина этого заключается в том, что он оставлял бы информацию в стеке намного дольше, чем необходимо, удаляя ее только при выполнении возврата. Фактически в процессе построения каждого отдельного вычисления никогда не происходило бы какого-либо уплотнения стека, даже если большая часть хранимых в стеке данных становилась бы излишней вскоре после помещения их туда. В этом разделе описываются способы преодоления указанного недостатка и, стало быть, ограничения роста стека.

VII.3.1. Восстановление пространства после успешного выхода из процедуры

Когда управление осуществляет успешный выход из процедуры, единственной информацией, которая касается текущего вычисления и которую нужно сохранить из всего того, что было порождено с момента входа в процедуру, является лишь множество значений данных, присвоенных переменным из обращавшегося к ней вызова. Поэтому всякий раз, когда происходит успешный выход из процедуры, появляется значительный простор для удаления информации из стека. Для того чтобы яснее понять, что сюда относится, рассмотрим программу, среди про-

цедур которой встречаются следующие (здесь указаны лишь имена процедур).

Программа 41 (фрагмент)

```
P1 : A если B,C
P2 : B если G
P3 : C если D,E
P4 : D если F
P5 : E
P6 : F
P7 : G
```

На рис. VII.8 изображен сегмент стека, в котором содержатся фреймы, образованные в ходе решения вызова **A**, т. е. начиная

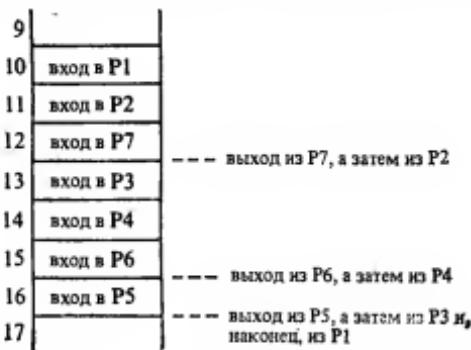


Рис. VII.8. Фреймы, порожденные в результате решения вызова **A**.

с того момента, когда управление входит в процедуру **P1**, и до момента его успешного выхода из нее. Для удобства ссылок позиции фреймов в стеке произвольным образом были занумерованы — от 10 до 16. Результатом решения вызова **A** является присваивание данных его переменным, все ячейки которых расположены в фреймах, предшествующих фрейму 10. Эти данные накапливаются в процессе образования фреймов 10—16. Когда вызов становится решенным, указанные фреймы — при выполнении двух описываемых ниже условий — могут быть удалены из стека (точнее, на их место могут быть записаны новые фреймы). Мы говорим в этом случае, что их пространство в стеке было *восстановлено*.

На практике за один раз в стеке восстанавливается пространство лишь одного фрейма — каждый выход из процедуры инцинирует (если это возможно) восстановление фрейма, образованного при входе в эту процедуру. В рассматриваемом примере при выходе управления из процедуры **P1** фрейм 10 может

быть восстановлен только при выполнении следующих двух условий:

- (а) в ячейках фреймов, предшествующих фрейму 10, не содержится указателей на какие-либо ячейки переменных в нем (т. е. на ячейки переменных из процедуры Р1);
- (б) ближайшая точка возврата расположена раньше фрейма 10.

В самом деле, допустим, напротив, что фрейм 10 был удален при нарушении условия (а). Для выполнения последующей унификации может потребоваться найти значение одной из ячеек в более раннем фрейме и возможна такая ситуация, когда этим значением является указатель на какую-то ячейку из фрейма 10. А так как в силу нашего предположения он уже был удален, то указатель в этом случае «повиснет» (его имеющегося в виду референта больше не существует) и, стало быть, процесс исполнения программы нарушится.

С другой стороны, допустим теперь, что фрейм 10 был удален, и при этом условие (а) выполнялось, а условие (б) было нарушено. В ходе последующего возврата при поисках новых путей решения вызова А в следе могли бы встретиться ссылки на некоторые из переменных процедуры Р1, требующие вернуть (теперь уже отброшенные) ячейки этих переменных в состояние *, что также нарушило бы процесс исполнения.

Условие (б) в период исполнения проверяется легко — для этого нужно лишь просто посмотреть содержимое регистра БТВ. Однако условие (а) во время исполнения нельзя проверить, не затрачивая на это слишком много времени. В силу этой причины в некоторых реализациях используется более сложное представление состояния исполнения, которое, как мы увидим в следующем разделе, гарантирует, что условие (а) всегда выполняется.

VII.3.2. Двухстековое представление

Двухстековое представление предназначено для того, чтобы устраниить определенные указатели, которые в противном случае после работы механизма восстановления могли бы остаться повисшими. В стандартной системе с совместным использованием структур подобные указатели имеют довольно специфический вид. Допустим в качестве примера, что в вызове А, обращающемся в момент образования фрейма 10 к процедуре Р1, одним из фактических параметров является переменная *x*, которой еще не было присвоено никакого значения. Это означает, что в некотором фрейме, предшествующем фрейму 10, переменной *x* уже будет выделена ячейка, находящаяся в рассматривае-

мый момент в состоянии *. Допустим далее, что соответствующим формальным параметром в заголовке Р1 является структурированный терм $f(y, z)$. При образовании фрейма 10 в результате обращения вызова А к процедуре Р1 должно (посредством унификации) получиться присваивание $x := f(y, z)$. Как мы уже видели, в одностековых системах с совместным использованием структур это присваивание представляется путем хранения в ячейке переменной x молекулы, указатель среды в которой указывает на фрейм 10. Вот этот указатель и относится как раз к тому виду указателей, у которых появлялся бы риск повиснуть, если бы в дальнейшем после выхода из Р1 фрейм 10 был удален: указатель такого вида используется для представления выходного присваивания структурированных данных из процедуры, чей входной фрейм мы хотим удалить после успешного выхода из нее.

Двухстековое представление было разработано Д. Уорреном в Эдинбургском университете для реализации Пролога на машине DEC-10. В этом представлении проблема повисающих указателей преодолевается путем использования вспомогательного стека, называемого часто *глобальным стеком*, для размещения ячеек тех переменных, которые встречаются в структурированных термах входной программы. В только что рассмотренном примере ячейки переменных y и z из процедуры Р1 помещались бы тогда в фрейме, расположением в глобальном стеке, и указатель среды из ячейки переменной x отсылал бы именно к этому фрейму, а не к фрейму 10.

Общая организация двухстековых систем, следовательно, такова. Основой стек, называемый *локальным стеком*, очень похож на те, которые используются в одностековых системах. В каждом из его фреймов содержатся обычные ячейки управления. Кроме того, в нем содержатся ячейки только для тех переменных из вызываемой процедуры, на значения которых никогда не могли бы ссылаться ячейки, расположенные в более ранних локальных фреймах. В каждом глобальном фрейме из глобального стека содержатся ячейки для всех оставшихся переменных из вызываемой процедуры. Таким образом, эта схема основана на разделении переменных, встречающихся во входной программе: они классифицируются либо как локальные, либо как глобальные. При входе в процедуру всегда образуется локальный фрейм. Если процедура содержит глобальные переменные, то образуется также глобальный фрейм, который связывается со своим локальным напарником соответствующим указателем, храящимся в локальном фрейме. Более точно, в локальном фрейме выделяется шестая ячейка управления, куда помещается *указатель глобального фрейма ($\Gamma\Phi$)*, указывающий на его глобального напарника. Эти два фрейма совместно опре-

деляют среду для вызовов из тела рассматриваемой процедуры. Выполнение процесса возврата приводит к обычному сокращению локального стека, причем всякий раз, когда в ходе возврата удаляется локальный фрейм, то удаляется и его глобальный напарник (если, конечно, он имеется). На рис. VII.9 показана связь локального и глобального стеков.

Итак, при двухстековом представлении допустимое содержимое ячеек, предназначенных для переменных из процедур, мож-

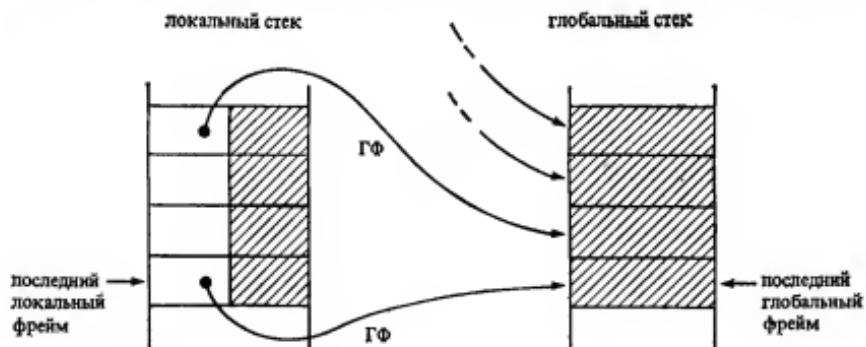


Рис. VII. 9. Локальный и глобальный стеки (заштрихованные клетки — ячейки переменных, пустые — ячейки управления).

но описать тогда следующим образом. Обозначим локальные и глобальные фреймы, в которых расположены ячейки этих переменных, буквами L и G соответственно. Тогда

(i) каждая ячейка в L или G может содержать либо символ $*$ (означающий, что переменной еще не присвоено никакого значения), либо константу, либо указатель на терм без переменных, хранящийся во входном массиве данных;

(ii) каждая ячейка в L может указывать на ячейку другой переменной (присваивание одной переменной значения другой) при условии, что последняя расположена либо в каком-то глобальном фрейме, либо в каком-то локальном фрейме, появившемся не позднее фрейма L ;

(iii) каждая ячейка в L может содержать молекулу при условии, что ее указатель среди отсылает либо к какому-то глобальному фрейму, либо к какому-то локальному фрейму, появившемуся не позднее фрейма L ;

(iv) каждая ячейка в G может содержать указатель на какую угодно другую глобальную ячейку (присваивание одной переменной значения другой) или она может содержать молекулу, указатель среди которой отсылает к какому-то глобальному фрейму.

Совместное действие этих ограничений заключается в том, что независимо от унификаций, выполнявшихся при входе в процедуру Р1 или после него, ни глобальные ячейки, ни ячейки локальных фреймов, предшествующих Л, никогда не могут указывать ии на фрейм Л, ни на ячейки, в ием содержащиеся. Другими словами, когда управление выходит из Р1, условие (а) удаления локального фрейма 10 обязательно выполняется. Сами эти ограничения возникают отчасти из-за критерия, применяемого для того, чтобы отличать локальные переменные от глобальных, отчасти из-за трактовки присваивания одиой переменной значения другой (в соответствии с которой значение переменной из вызова всегда присваивается переменной из заголовка процедуры) и отчасти из-за принятого соглашения, в силу которого все указатели между локальным и глобальным стеками неизменно отсылают из первого ко второму.

VII.3.3. Механизм восстановления при успешном выходе из процедуры

Удовлетворение условия восстановления (а) зависит от соответствующих механизмов, реализуемых для выполнения унификации, и от распределения локальной и глобальной памяти. Будет или нет выполняться второе условие (б) зависит от текущего состояния регистра БТВ: оно выполняется, если только БТВ есть \perp или указывает на некоторый фрейм, предшествующий фрейму 10 (т. е. тому фрейму, который восстанавливается). Реализацию этой проверки, а также механизма восстановления можно осуществить путем следующего простого преобразования шага 2 в приведенном на рис. VII.3 алгоритме управления.

Шаг 2 (Выбор вызова)

```

if ТП → есть факт
then TB := П(f)
    if BTB = ⊥ или BTB < f then f := f - 1
    while TB = выход и BT ≠ 1
        do if BTB = ⊥ или BTB < БП then f := f - 1
            TB := П(БП)
            БП := РФ(БП)
    if TB = выход
    .
    .
далее точно так же, как и прежде

```

Следствием проведенной модификации является то, что всякий раз, когда происходит выход из процедуры (быть может, внутри цепочки выходов, осуществляемых циклом на шаге 2) и удов-

летьворяются условия (а) и (б) восстановления локального фрейма, образованного при входе в процедуру, этот фрейм обязательно будет находиться в данный момент на вершине стека, и его можно сразу же удалить, уменьшая с этой целью регистр f позиции вершины локального стека ($f := f - 1$).

Когда описанный алгоритм применяется к исполнению программы 41, в результате выходов из процедур P7 и P2 будут восстановлены соответственно позиции 12 и 11, так что два следующих фрейма (для процедур P3 и P4) займут тогда позиции 11 и 12, а не 13 и 14. В дальнейшем аналогичные восстановления произойдут в результате выходов из процедур P6, P4, P5 и P3. Наконец, последующий выход из процедуры P1 восстановит фрейм, образованный при входе в P1, и, таким образом, в течение всего процесса решения вызова А локальный стек никогда не будет продолжаться дальше позиции 13. Единственное, что сохранится в результате всего этого процесса после восстановления фрейма 10 — это данные, присвоенные переменным из вызова А, если, конечно, такие присваивания были сделаны.

Отметим, что описанный механизм никогда не приводит к сжатию глобального стека, и поэтому данные, хранящиеся в этом стеке, имеют гораздо более длительное время жизни, чем те, которые хранятся в его локальном аналоге. Это свойство возлагает дополнительное бремя на разработчика реализации, поскольку в ходе исполнения программы данные из глобального стека могут стать излишними. Само по себе вреда это неносит, однако если глобальному стеку грозит переполнение и в нем содержится много излишних данных, то, может быть, интерпретатору имеет смысл запустить программу *сборки мусора*, которая обнаруживает эти данные и отбрасывает их. Хорошо написанная программа, предназначенная для выполнения сборки мусора, будет поглощать много времени, если неудачно выбрана стратегия ее запуска. Более того, составление подобной программы само по себе представляет значительный программистский проект.

Какие бы средства для сборки мусора в глобальном стеке ни использовались, ясно, что они должны в первую очередь поддерживаться хорошей стратегией определения локального или глобального статуса переменных. Поскольку восстановление локального стека операционо несложно и (обычно) выполняется довольно часто, желательно, очевидно, предоставить локальный статус по возможности большему числу переменных. Однако задачу, состоящую в том, чтобы точно определить в период исполнения, какие из переменных *должны* быть глобальными, нельзя полностью решить в момент входа в процедуру. В принципе интерпретатор может сделать в этот момент временное распределение ячеек по локальному и глобальному фреймам,

которое хотя и не будет оптимальным, но окажется достаточным для того, чтобы предотвратить в дальнейшем возникновение потенциально повисающих указателей. Впоследствии в процессе вычисления можно будет перераспределить эти ячейки (и соответствующим образом внести поправки в ссылки на них) в свете выполнявшихся присвоений, но это потребует много дополнительного времени и может все еще не дать оптимального результата. Поэтому разделение встречающихся в процедуре переменных на локальные и глобальные происходит обычно в период компиляции, и в основу его положено простое синтаксическое правило: переменным выделяются глобальные ячейки в том и только том случае, когда они имеют вхождения в какие-либо структурированные термы этой процедуры. Данное правило не является оптимальным, поскольку в соответствии с ним некоторые переменные в общем случае делаются глобальными, даже если предоставление им вместо этого локального статуса не нарушило бы последующего исполнения программы. С другой стороны, применение этого правила обходится дешево, оно дает всем входным фреймам для каждой конкретной процедуры одну и ту же структуру независимо от контекста, в котором происходит вызов этой процедуры в период исполнения, и оно гарантирует, что условие восстановления (а) будет всегда выполняться.

VII.3.4. Оптимизация последнего вызова

Восстановление локального входного фрейма процедуры не всегда нужно откладывать до тех пор, пока управление осуществит успешный выход из процедуры. При выполнении некоторых условий восстановление этого локального фрейма (но не его глобального напарника) может в действительности иметь место в тот момент, когда активируется *последний вызов* из процедуры. Другими словами, локальный фрейм иногда можно восстанавливать, как только интерпретатор начнет свою попытку решить этот вызов, а не (как это было раньше) после того, как он успешно его решит. Такая тактика называется *оптимизацией последнего вызова*, и ее применение может существенно улучшить как использование памяти, так и скорость обработки данных.

Рассмотрим снова программу 41 вместе с рис. VII.8 и допустим, что управление только что вышло из процедуры P2. Допустим, кроме того, что входные фреймы для процедур P2 и P7 были удалены либо посредством применения стандартного механизма восстановления при успешном выходе, либо посредством оптимизации последнего вызова. Текущей позицией вершины стека в нашем локальном стеке является, следовательно,

позиция 10, на которой в данный момент расположен локальный входной фрейм процедуры Р1; в дальнейшем ради краткости мы будем называть этот фрейм *удаляемым фреймом*. Поскольку следующая задача интерпретатора заключается в активации последнего вызова С из Р1, складывается ситуация, в которой можно попробовать применить оптимизацию последнего вызова (короче, ОПВ). Если выполняется ОПВ, то ее результатом будет запись следующего образуемого локального фрейма на месте удаляемого, что, как и требовалось, равносильно восстановлению пространства последнего. Следующим фреймом будет локальный входной фрейм для процедуры Р3, т. е. для той процедуры, которая отвечает на последний вызов из Р1.

Для выполнения ОПВ необходимо, чтобы соблюдались следующие условия:

- (с) удаляемый фрейм не должен являться точкой возврата;
- (д) точкой возврата не должен являться также новый локальный фрейм, который будет записан на месте удаляемого;
- (е) новый локальный фрейм не должен содержать указателей на удаляемый фрейм.

Эти условия как по форме, так и по мотивировке аналогичны условиям (а) и (б) восстановления после успешного выхода из процедуры. По существу они предотвращают появление ссылок на удаляемый фрейм в ходе дальнейшего исполнения программы. Первые два условия можно проверить в период исполнения, соответствующим образом учитывая состояние регистра БТВ. Выполнение последнего условия (е) в системах с совместным использованием структур можно гарантировать, производя в период компиляции соответствующие распределения по локальным и глобальным фреймам, как это объясняется ниже.

При образовании на следующем шаге (соответствующем входу в Р3) локального и глобального фреймов выполняется унификация, которая, вообще говоря, должна учитывать среду последнего вызова С, содержащуюся в локальном и глобальном фреймах, образованных при входе в Р1. Эта унификация может, кроме того, породить данные для переменных из нового локального фрейма, который должен в конечном счете занять пространство стека, в настоящее время занимаемое удаляемым фреймом. Поэтому для того, чтобы предотвратить конфликт по совпадению обращений для чтения и записи (т. е. запись данных в ту область памяти, изначальное содержимое которой могло бы еще понадобиться для использования в дальнейшем), целесообразно сначала скопировать отобранные из удаляемого фрейма данные во временных регистрах. В этом случае возможно выполнить целый ряд оптимизаций очень низкого уровня (здесь не детализируемых), которые сводят до минимума поток данных между

упомянутым регистрами, старыми ячейками и новыми ячейками и тем самым приводят к значительному сокращению времени обработки, расходуемому как на унификацию, так и на построение фрейма. Интересная проблема встает перед разработчиком реализации в связи с удовлетворением условия (e). Одни из подходов к ее решению заключается в том, чтобы просто заставить интерпретатор исследовать результаты выполнявшейся на новом шаге унификации и затем определить, можно ли это осуществлять ОПВ. Для этого, однако, может потребоваться слишком много времени, что частично нарушит цели оптимизации. С другой стороны, можно попытаться прежде всего предотвратить возникновение неугодных указателей с помощью принятия соответствующих мер предосторожности в период компиляции.

Для того чтобы понять, откуда могли бы возникнуть такие указатели, рассмотрим задачу унификации некоторого параметра $T1$ из (входной формы) вызова C в процедуре $P1$ с соответствующим формальным параметром $T2$ из заголовка вызываемой процедуры $P3$. Если параметр $T1$ не является переменной, то все содержащиеся в нем переменные обязательно должны быть распределены по глобальным ячейкам; следовательно, все указатели на них, порожденные в ходе унификации, после ликвидации удаляемого фрейма повиснут не могут. Если параметр $T2$ не является переменной, а $T1$ — некоторая переменная x , то их унификация даст присваивание $x := T2$, и какой бы вид ни имел $T2$, указатель на ячейку x в результате этого никогда не возникнет. Приведенные доводы сужают область наших рассмотрений до случая, когда параметр $T1$ есть переменная x , а параметр $T2$ — переменная z , т. е. когда имеет место следующая ситуация

$$\begin{aligned} P1 : & A(\dots) \text{ если } B(\dots), C(\dots, x, \dots) \\ P3 : & C(\dots, z, \dots) \text{ если } D(\dots), E(\dots) \end{aligned}$$

Последний вызов $C(\dots, x, \dots)$ из процедуры $P1$ активизируется в контексте соответствующей ему среды, и она будет содержать ячейку переменной x . Рассмотрим теперь четыре случая, исчерпывающие все те ситуации, которые могут возникнуть непосредственно перед выполнением унификации.

(1) Допустим, что x — глобальная переменная (имеющая вхождения в структурированный терм); тогда, в силу соглашений относительно указателей между локальным и глобальным стеками, никакой указатель, помещаемый в ее ячейку в результате выполнения унификации, не может отсылать к удаляемому фрейму.

(2) Допустим, что x — локальная переменная, а ее ячейка содержит некоторое значение t (уже присвоенное x), не указы-

вающее ни на удаляемый фрейм, ни на какую-либо его ячейку; тогда присваивание $z := t$, полученное в результате выполнения унификации, не может образовать указателя на удаляемый фрейм.

(3) Допустим, что x — локальная переменная и ей еще не присвоено никакого значения; тогда полученное в результате выполнения унификации присваивание $z := x$ (заметим, что присваивание $z := *$ не годится) помещают в ячейку переменной z потенциально повисающей указатель на ячейку x в удаляемом фрейме.

(4) Допустим, что x — локальная переменная и присвоение ей значения содержит указатель на удаляемый фрейм или какую-либо его ячейку. Им мог бы быть только указатель на ячейку некоторой другой переменной y из процедуры P1 (этот указатель не мог бы быть молекулой, поскольку ее указатель среды отсылал бы к глобальному фрейму); таким образом, если переменная y не является глобальной, то получаемое в результате выполнения унификации присваивание $z := y$ помещает в ячейку переменной z потенциально повисающей указатель на ячейку y в удаляемом фрейме.

Из этого анализа вытекает, что для безопасного выполнения ОПВ следует устранить случаи (3) и (4), в каждом из которых переменная x является локальной. Средство для достижения указанной цели, предложенное Д. Уорреном и реализованное в его системе DEC-10 Пролог, заключается в предоставлении переменной x глобального статуса, если x не имеет вхождений в заголовок процедуры P1. В этом случае всякий указатель на ячейку x , который в результате выполнения унификации мог бы быть присвоен ячейке z , никогда не будет повисшим, и, стало быть, ОПВ можно осуществлять, не причиняя ущерба. В противном случае, если x является локальной и имеет вхождения в заголовок процедуры P1, то стандартное соглашение относительно унификации гарантирует, что при входе в P1 переменной x сразу же будет присвоено какое-то значение, и поэтому случай (3) невозможен.

Рассмотрим, наконец, случай (4). В его допущениях требуется, чтобы присваивание $x := y$ имело место до того, как будет активирован вызов С. Подобное присваивание никогда не могло бы произойти при входе в процедуру P1. Следовательно, оно должно быть выполнено в результате решения вызова В, для чего в свою очередь потребовалось бы, чтобы после входа в P1 переменной x не было присвоено никакого значения (поскольку одной и той же переменной не могут быть присвоены два значения). Это означало бы, что x не имеет вхождений в заголовок процедуры P1. Таким образом, случай (4) возникает

только тогда, когда переменная x является локальной и не имеет вхождений в заголовок процедуры — возможность, которая устраняется с помощью предложенного Уорреном средства.

Итак, мы получаем, что для того, чтобы гарантировать выполнение условия (e), достаточно дополнить используемые в период компиляции обычные правила разделения встречающихся в Р1 переменных на локальные и глобальные еще одним новым простым правилом. Для выполнения условия (c) требуется только, чтобы регистр BTB перед тем, как произойдет новый шаг, либо находился в состоянии -1 , либо указывал на какую-то позицию, предшествующую удаленному фрейму. Для выполнения условия (d) требуется только то, чтобы новый шаг не приводил к появлению ненспробованных кандидатов ($SCA = -1$) после того, как будет выбрана очередная вызываемая процедура (Р3).

Когда имеет место ОПВ, указатель (перехода) П из удаляемого фрейма в принципе будет утрачен вместе со всем остальным содержимым этого фрейма. Последующая операция перехода, которая предвосхищается указателем П, останется осуществимой при условии, что в новый фрейм будет помещен именно этот указатель перехода, а не (как было раньше) позиция успешного выхода из процедуры, расположенная вслед за оптимизируемым последним вызовом. Это предполагает, что реализация ОПВ должна сопровождаться более общей перестройкой алгоритма управления, так чтобы указатель перехода в каждом фрейме всегда указывал непосредственно на следующий ожидающий решения вызов в программе, и никогда не указывал на успешный выход из процедуры, разве только на успешный выход из целевого утверждения. Такая организация значительно улучшает скорость и непрерывность выбора вызовов, поскольку она устраниет неэффективность, связанную с прослеживанием цепочек успешных выходов. Однако результатом рассматриваемой модификации является также то, что содержащийся в фрейме указатель перехода не будет в этом случае достаточным базисом (как было раньше) для определения вызова, который был активирован при образовании фрейма. Тем не менее данная информация могла бы потребоваться, если бы фрейм оказался точкой возврата, так как в дальнейшем после выполнения некоторой операции возврата этот вызов нужно было бы определить для того, чтобы активировать еще раз. Из приведенных соображений вытекает, что в каждом фрейме (или по крайней мере в каждой точке возврата) должен быть также явно представлен еще в одной ячейке управления идентификатор вызова, обращавшегося к соответствующей данному фрейму процедуре. Тот, кто предполагает заняться реализацией и кто надлежащим образом понял механизм управления эле-

ментарного интерпретатора, не должен столкнуться с какими-либо трудностями при осуществлении всех этих модификаций, необходимых для обеспечения ОПВ.

Стоит отметить, что ОПВ восстанавливает явно пространство, занятое ненужными ячейками управления и ячейками локальных переменных, однако никогда не приводит к сокращению глобального стека, где накапливаются структурированные данные — он сокращается лишь при выполнении возврата или сборке мусора. С другой стороны, если ОПВ запрограммирована настолько компактно, насколько это возможно для того, чтобы избежать чрезмерного потока данных между фреймами и регистрами, то ОПВ может значительно сократить время обработки, поглощаемое итеративными вычислениями. В программе 40, например, такое вычисление получается в результате неоднократного вызова (детерминированной) итеративной процедуры **обратить***. Воздействие ОПВ на исполнение этой программы заключается в том, что на месте каждого фрейма, за исключением первого, записывается его последователь, и поэтому локальный стек никогда не продолжается дальше позиции 2, тогда как прежде его длина росла пропорционально длине входного списка. Однако второе возможное улучшение — это гораздо более высокая скорость, поскольку разработчик реализации может (после достаточного анализа и программистских усилий) «стянуть» унификацию и построение фрейма, возникающих при выполнении каждой операции перезаписи. Получаемое в результате поведение будет во многом подобно поведению традиционной программы, использующей деструктивное присваивание для проведения итерации над одной фиксированной областью памяти и в то же время строящей структурированные выходные данные в другой области; в логической реализации этими областями являются соответственно локальный и глобальный стеки.

Заметим, что при двухстековой реализации программы **обратить*** все изображенные на рис. VII. 7 ячейки переменных i_i , x_i и w_i будут находиться в фреймах из глобального стека, поскольку все переменные i , x и w из процедуры Р2 имеют вхождения в структурированные термы. Однако, как показано на рисунке, ни в одну из этих ячеек никогда не помещаются потенциально повисающие указатели, так что, строго говоря, ни одной из них не требуется находиться в глобальном стеке. Приведенный пример показывает, как в результате осуществляемого в период компиляции простого синтаксического управления статусом переменных глобальный стек в общем случае перегружается, что является платой за предоставление интерпретатору возможности безнаказанно восстанавливать все, что остается в локальном стеке. Вследствие этого утверждалось, что ОПВ как средство

экономии пространства не будет давать особого эффекта, если ее не дополнить либо сборкой мусора, либо более приемлемыми схемами разделения переменных.

VII.4. Экономия времени обработки данных

По сравнению со скоростью исполнения, которая достигается при написании программ на языках ассемблера или на языках относительно низкого уровня, подобных Бейсику и Фортрану, исполнение эквивалентных логических программ обычно происходит медленнее. До некоторой степени это обусловлено механизмами, необходимыми для обеспечения недетерминированного исполнения, однако в большей мере это является результатом недостаточной разработки методов эффективного манипулирования данными. В ранний период логического программирования при осуществлении реализаций много усилий было затрачено на усовершенствование обработки представлений посредством структурированных термов, в результате чего гораздо меньше внимания уделялось разработке мощных механизмов управления, предназначенных для манипулирования большими, нерегулярными структурами данных, представленных иными способами.

Вероятно, эта ситуация изменится, как только интерес к логическому программированию, первоначально концентрировавшийся в академических кругах, связанных с искусственным интеллектом, где знакомство с родственными формальными системами, подобными Лиспу, — вторая натура, постепенно распространится и на другие области, такие как научная и коммерческая обработка данных. Требования, предъявляемые этими сферами применения, приведут, без сомнения, к изменению стиля логического программирования и, следовательно, технологии его реализации — впрочем, в такой же степени они и сами изменятся под его воздействием.

Эволюция технического обеспечения новых ЭВМ от последовательных управляемых процессами машин фон Неймана по направлению к многопроцессорной архитектуре, обеспечивающей параллелизм, режимы потока данных и встроенные механизмы (такие как сопоставление с образцом) и разработанной специально для нужд формализмов, основанных на построении вывода, также внесет радикальные изменения в способ реализации логики как языка программирования.

В силу всех этих причин не следует ожидать, что рекомендации использовать какие-либо предположительно оптимальные стратегии обработки данных останутся справедливыми даже в течение непродолжительного времени. Поэтому в данном раз-

деле описывается лишь небольшая выборка имеющихся у разработчика реализаций возможностей оказывать влияние на скорость обработки в контексте существующих на сегодняшний день стиля программирования и машинной архитектуры.

VII.4.1. Системы без совместного использования структур

Все логические реализации можно, вообще говоря, разделить на два класса: системы, основанные на совместном использовании структур (CC-системы), и системы, в которых этот метод не используется (HCC-системы). Мы дадим здесь краткое описание HCC-системам, а затем проведем сравнение двух упомянутых подходов.

В HCC-системах, которые в настоящее время составляют меньшую часть от существующих реализаций, используются, как правило, две основные динамические области памяти. Первая из них представляет собой *локальный стек*, который фактически соответствует локальному стеку в CC-системах за исключением того, что в каждом фрейме помимо обычных управляющих ячеек предусмотрены локальные ячейки для всех различных переменных из вызываемой процедуры. Второй стек играет вспомогательную роль: в него помещаются явные (или «коикретные») представления структурированных термов, присвоенных (посредством унификации) в качестве значений локальным переменным. Организация этой области памяти более подобна неупорядоченному массиву, чем стеку, однако обычно ее называют *глобальным стеком* (или «стеком копий»).

Допустим в качестве примера, что активируется некоторый вызов **обратить*($t, z, v.v.z$)**, причем переменным t, z и v еще не присвоено никаких значений, и пусть он обращается к нашей процедуре

P2 : обратить*($x, u.w, y$) если обратить*($u.x, w, y$)

В типичной HCC-системе получающиеся в результате этого присваивания $\{x := t, z := u.w, y := v.v.z\}$ могли бы тогда быть представлены так, как показано на рис. VII.10. Ячейки переменных t, z и v , изображенные на этом рисунке, находятся в различных локальных фреймах, образованных на предыдущих шагах. В новом локальном фрейме, образованном при входе в процедуру P2, размещаются ячейки переменных x, y, u и w . Кластер ячеек справа представляет собой новое пополнение глобального стека и является явным представлением терма $v.v.u.w$. Указатели на него из ячеек для y и z означают соответственно присваивания $y := v.v.z$ и $z := u.w$. Заметим, что цепочка указателей связывает локальную ячейку переменной v с двумя ассоциированными глобальными ячейками в кластере. Таким

образом, в НСС-системе используются в общей сложности три ячейки для v , тогда как в СС-системе использовалась бы лишь одна (глобальная) ячейка. Если на каком-то последующем шаге переменной v будет присвоен в качестве значения некоторый структурированный терм, то этот терм в свою очередь помещается в явном виде в глобальный стек, а указатель на него будет храниться в четвертой ячейке рассматриваемого кластера.

Главное различие между СС- и НСС-системами состоит в том, что в последней присваивание структурированного терма

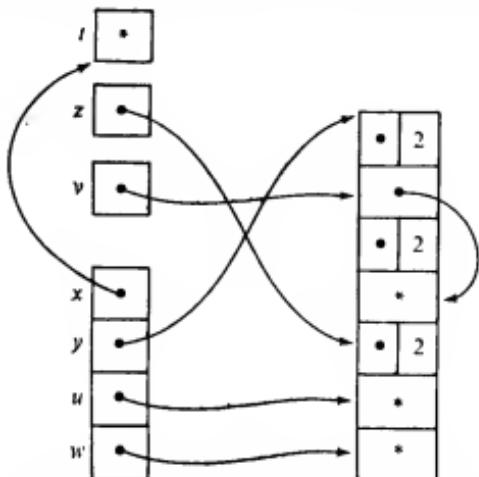


Рис. VII. 10. Использование локальных и глобальных ячеек в НСС-системах.

переменной представляется просто путем соединения (с помощью указателя) локальной ячейки этой переменной с глобальным кластером, содержащим компактную явную копию упомянутого терма, в то время как в СС-системе терм представляется более разбросанно в виде иерархии скелетов «чистого кода», которые требуется интерпретировать в контексте различных сред присваиваний, раскиданных по двум стекам. Стало быть, НСС-система, в отличие от СС-систем, обычно обеспечивает по существу прямой доступ (т. е. минимальное использование указателей) к уже вычисленным ею структуризованным данным. Фактически, совместное использование структур можно рассматривать как «ленивый» метод построения данных. При представлении множества связанных друг с другом присваиваний вида

$$\{x_i := f(y_j), y_j := g(z_k), z_k := NIL\}$$

порожденных в результате выполнения унификации на различных шагах i, j и k , в СС-системе не делается никакой попытки

построить явные (полностью разыменованные) значения $g(NIL)$ и $f(g(NIL))$ переменных y_i и x_i . Напротив, единственным средством для построения этих явных значений, потребуйся он когда-либо (например, для сообщения о полученном решении целевого утверждения), в СС-системе являются указатели или молекулы, которые хранятся в ячейках переменных. Кроме того, в СС-системе часто приходится выбирать компоненты структурированных термов, когда осуществляется попытка унифицировать эти термы с какими-то другими. При такого рода обстоятельствах мы фактически оказываемся вынужденными использовать в реализации операции построения низкого уровня, чего ранее избегали.

Эта оценка должна быть уравновешена тем наблюдением, что при исполнении многих программ порождаются структуры, которые в дальнейшем не участвуют ни в получении разыменованных выходных данных, ни в попытках выполнения унификации. Поэтому, выбирая между СС- и НСС-системами, следует рассмотреть вопрос о том, будет ли в предполагаемых исполнениях программ преобладать выборка структурированных данных или просто их построение. Вообще говоря, для последней категории исполнений больше подходят СС-системы, поскольку задача, состоящая в построении молекул, оказывается тривиальной (решается очень быстро) по сравнению с задачами их разыменования или выборки в ходе выполнения унификации их глубоко расположенных компонент.

Кроме того, при выборе должно учитываться потенциально малое потребление памяти в СС-системах. Эту особенность, однако, не следует переоценивать: сравнительные исследования показали, что при отсутствии сборки мусора в глобальном стеке СС-системы вполне могут потреблять в целом больше памяти, чем ее расходовалось бы в НСС-системах. Это происходит потому, что, согласно используемой в СС-системах стратегии распределения переменных по стекам в период компиляции — всегда неоптимальной, а иногда оказывающейся значительно хуже оптимальной, — пространство глобального стека представляется тем переменным, которые, исходя из чисто синтаксических соображений, могли бы стать компонентами структурированных термов, вычисленных в ходе исполнения программы. В противоположность этому в НСС-системах пространство глобального стека распределяется в период исполнения, и представляется оно только тем структурированным термам, которые уже *действительно* вычислены. В любом случае оба подхода являются неоптимальными, если в ходе исполнения программы некоторые глобальные данные становятся излишними, поскольку в обычных процессах восстановления локального стека (исключая процесс возврата) они не рассматриваются.

В приложениях, где, по всей видимости, отдельные вычисления будут занимать намного меньше памяти, чем ее физически имеется, относительные требования, предъявляемые к объемам памяти в СС- и НСС-системах, могут почти не иметь значения (часто различаясь между собой лишь на 10—50 %). Если это действительно так, то выбор системы должен в большей степени основываться на сравнении ожидаемой скорости обработки данных. Любая всесторонняя оценка этого параметра обязана учитывать множество факторов, включающее характеристики входных программ, связанные с выборкой данных и их построением, возможности оптимизации процесса унификации, а также адресное пространство вычислительной машины, на которой осуществляется реализация, структуру ее слов и механизмы адресации.

VII.4.2. Компиляция

В большинстве существующих реализаций логики как языка программирования программы исполняются в режиме интерпретации. По этой причине особенно благоприятные условия создаются для такой разработки программ, когда у пользователя часто возникает желание изменить или повторно прогнать свои программы, не сталкиваясь с утомительными и поглощающими время требованиями повторной компиляции. Кроме того, в условиях интерпретации легче включать и использовать средства трассировки и отладки.

С другой стороны, компиляция (трансляция) логической программы в непосредственно исполняемые машинные коды предоставляет в принципе гораздо большие возможности для достижения ее эффективного исполнения. Главная причина этого заключается в том, что интерпретатор должен основываться на единственной общей программе унификации, способной обрабатывать любые предлагаемые ей пары (вызов, заголовок процедуры). Как правило, однако, чем более общей является программа, тем менее эффективной она оказывается в конкретных ситуациях, поскольку в ней всегда должны быть предусмотрены средства для обстоятельств, которые на самом деле могут не возникнуть. Компилятор может обходиться без такой общей программы, а вместо этого образовывать для каждого заголовка входной процедуры сегмент унификации, представленный в машинных кодах и специально приспособленный для формальных параметров, входящих в заголовок. Основная система исполнения состоит тогда из управляющей программы, входных процедур вместе с их заголовками, скомпилированными в машинные коды, и каких-либо библиотечных программ, также в виде скомпилированных кодов.

Единственные имеющиеся в настоящее время полностью действующие логические компиляторы разработаны Дэвидом Уорреном для машины DEC-10. Эффективность их работы, связанной с образованием эффективных объектных программ, значительно повышается благодаря добавлению к входным программам деклараций видов. В этих декларациях объясняется ожидаемый вид фактических параметров, которые будут переданы процедурам в ходе исполнения программы. Так, например, компиляцию программы 40 можно сделать более эффективной, добавляя к ней декларацию

вид обратить*(+, +, -)

в которой объявляется, что всякий раз, когда активируется вызов обратить*, его первые два фактических параметра переменными не являются, однако третий параметр — переменная. Эти знания компилятор использует с целью специализации кода, образованного для заголовков процедур P1 и P2, который он эффективным образом оптимизирует, имея в виду предполагаемый способ его употребления.

Полезным методом реализации сложного логического компилятора, позволяющим избежать бремени написания самого компилятора в машинных кодах, является метод, называемый *раскруткой*. Сначала компилятор С пишется как множество Р логических процедур, описывающих все стандартные задачи компиляции, которые включают в себя ввод исходной программы и уплотнение, синтаксический разбор, построение таблицы идентификаторов, диагностику ошибок и генерацию машинной программы. Таким образом, при помощи процедур из Р можно решить любой вызов вида скомпилировать (*множество-процедур, соответствующий-код*). Затем составляется целевое утверждение G, которое в качестве выходных данных требует выдать код, соответствующий самому множеству процедур Р. Образованную в результате этого программу (Р, G) можно теперь исполнить с помощью какой-либо уже имеющейся логической реализации и тем самым получить на выходе версию компилятора С, представленную в машинных кодах, — иными словами, эффективный действующий компилятор для любого другого множества логических процедур.

VII.4.3. Унификация

Значительные возможности для повышения скорости исполнения логических программ заключены в реализации механизма унификации. Наиболее часто для выполнения унификации используются различные варианты алгоритма Робинсона. В контексте логического программирования этот алгоритм начинается

с помещения в списки L1 и L2 фактических параметров из активированного вызова и формальных параметров из заголовка выбранной процедуры соответственно. Третий список Θ служит для записи присваиваний, образованных в ходе согласования параметров из списков L1 и L2. В начальный момент в Θ фиксируется (каким-либо подходящим для последующего употребления образом) текущий статус переменных из указанных параметров, когда им не присвоено еще никаких значений. Если алгоритм завершается успешно, то требуемым наиболее общим унифициатором для данного шага является заключительное состояние списка Θ ; в противном случае унифициатора не существует.

Алгоритм представляет собой по существу один основной цикл, на каждом i -м шаге которого сравниваются между собой два выражения $t_1\Theta$ и $t_2\Theta$, где t_1 и t_2 — i -е элементы списков L1 и L2 соответственно. Как обычно, выражение $t\Theta$ означает здесь применение к терму t содержащихся в текущем состоянии списка Θ подстановок термов вместо переменных. Проверки, выполняемые на i -м шаге, и получаемые на нем результаты таковы.

1. (а) Если одно из выражений $t_1\Theta$ или $t_2\Theta$ является переменной, содержащейся в другом выражении, то алгоритм заканчивается неудачей: унификация невозможна.

(б) Если одно из выражений $t_1\Theta$ или $t_2\Theta$ является переменной, не содержащейся в другом выражении, то в список Θ заносится присваивание первому выражению (переменной) второго в качестве значения (тем самым Θ обновляется).

2. Если одно из выражений $t_1\Theta$ или $t_2\Theta$ является константой, а другое — либо константа, отличная от первой, либо структурированный терм, то алгоритм заканчивается неудачей: унификация невозможна.

3. (а) Если выражения $t_1\Theta$ и $t_2\Theta$ — структурированные термы, имеющие различные главные функции, то алгоритм заканчивается неудачей: унификация невозможна.

(б) Если оба выражения $t_1\Theta$ и $t_2\Theta$ — структурированные термы, имеющие одинаковые главные функции, то их аргументы добавляются в конце списков L1 и L2 соответственно (тем самым L1 и L2 обновляются).

Алгоритм успешно завершает работу только в том случае, когда выполнится сравнение всех элементов L1 и L2 и при этом не произойдет неудачного исхода.

Приведенное описание, конечно же, является лишь абстрацией того, что в действительности имеет место в конкретной реализации. Подлежащие сравнению выражения $t_1\Theta$ и $t_2\Theta$ обычно в явном виде не строятся. В частности, в системе с совместным использованием структур вид терма t_1 устанавливается лишь посредством анализа среды рассматриваемого вызова, а

также всех тех других сред, на которые в первой содержатся ссылки. Более того, в результате обновления списка Θ , выполнившегося в п. 1(b), значения могут быть присвоены ячейкам переменных как из этой среды, так и из предыдущей (при передаче выходных данных вызову); они могут быть присвоены также ячейкам переменных из новой среды, образуемой для текущего шага (при передаче входных данных процедуре). Эти возможности предоставляют разработчику реализации широкий простор для оптимизации решаемых программой унификации задач, связанных с обновлением различных сред в стеках и организацией доступа к ним. Так, например, одно из решений, которое нужно принять при разработке программы унификации, касается двух возможностей: следует ли обновлять соответствующие ячейки переменных, как только на некотором шаге цикла будет обновлен список Θ , или же вместо этого следует хранить присваиваемые значения во временных регистрах. В первом случае при неудачном исходе унификации потребуется определить все эти ячейки и вернуть их в прежнее состояние, тогда как при успешном ее завершении ячейки будут находиться именно в том состоянии, которое нам нужно. Во втором же случае при неудачном исходе они остаются в нужном состоянии (т. е. без изменений), в то время как после успешного завершения унификации полученные значения должны быть скопированы из регистров в соответствующие ячейки. Принятие такого рода решений еще больше усложняется при выполнении оптимизаций (таких как ОПВ), направленных на экономию памяти, когда один фрейм записывается на месте другого. В деталях многие подобные соображения описываются в исследовательской литературе.

Помимо проблем, связанных с обновлением ячеек переменных и организацией доступа к ним, главным препятствием быстрого выполнения унификации оказывается тест, отделяющий случай 1(a) от случая 1(b). Если одно из выражений является переменной, то с помощью этого теста требуется решить, имеет или нет упомянутая переменная вхождения в другое выражение. Его обычно называют *проверкой вхождения*, и служит он для того, чтобы предотвратить возникновение бессмысленных присваиваний вида $x := f(x)$. Часто приводится классический пример, в котором активируемым вызовом является $p(f(u), u)$, а заголовком процедуры — $p(x, x)$. Присваивания $\{x := f(u), u := x\}$, из которых вытекает $x := f(x)$, при последующих попытках разыменовать значение переменной x привели бы к бесконечному зацикливанию. Во многих реализациях проверка вхождения вообще не выполняется или делается необязательной на том основании, что нежелательные структуры параметров возникают редко и что принятие каких-либо мер предосторож-

ности необоснованно задержит выполнение большинства простых унификаций. В других же реализациях эта проверка опускается без всякого ущерба, поскольку в них допускается не более одного вхождения какой-либо переменной в заголовок процедуры. Подобное средство, однако, накладывает довольно неудовлетворительные ограничения на используемый стиль программирования.

Разработчик реализации может дополнительно сократить расходы, связанные с выполнением унификации в период исполнения, надлежащим образом воспользовавшись индексированием процедур, поскольку построение индексных таблиц в период компиляции является (неявным) частичным согласованием заголовка каждой процедуры с ожидаемым классом вызовов. Попытка унифицировать во время исполнения формальные параметры процедуры с известными фактическими параметрами начинается тогда с информации о том, что некоторое согласование уже было достигнуто благодаря выделению этой процедуры в качестве кандидата для вызова.

Программист может до произвольных пределов устраниТЬ бесполезные или какие-то иные нежелательные попытки выполнения унификации путем использования оператора отсечения (/), для реализации которого обычно требуется лишь простое обновление регистра БТВ и, быть может, нескольких ячеек ПТВ. Как правило, на операцию отсечения тратится гораздо меньше времени, чем экономится посредством устранения лишних неисprobованных кандидатов. Это справедливо, разумеется, при условии, что оператор отсечения используется в соответствии со здравым смыслом; при злоупотреблении им может произойти обратное.

VII. 5. Исторический очерк

Первая практическая реализация выдвинутой Колмероэ концепции языка Пролог как воплощения идеи Ковальского о логическом программировании была предпринята в 1972 г. Русселем в университете Экс—Марсель. Он применил метод совместного использования структур для резолютивного доказательства теорем, который незадолго до того был разработан Бойером и Муром (1972). Эта реализация была впоследствии описана в отчете Колмероэ и др. (1973). Написанная на Алголе-W, она оказалась медленной и расходовала чрезвычайно много памяти главным образом потому, что в ней излишне много внимания уделялось предвидению процесса возврата и отсутствовали механизмы восстановления. Затем Баттани и Мелони (1973) написали усовершенствованную версию Пролога на Фортране.

которая в дальнейшем была перенесена на несколько других вычислительных установок, в результате чего появилось целое семейство вариантов марсельского Пролога. Особенности и реализация версии, введенной в действие в Имперском колледже, описаны в магистерской диссертации Лихтмана (1975), а функционирование марсельского Пролога, с точки зрения пользователя, объясняется в справочном руководстве Русселя (1975).

К 1975 г. системы, подобные марсельской, написанные на CDL для ЭВМ ICL-1903A и системы 4/70, уже функционировали в Венгрии. Широкий спектр приложений этих, а также последующих версий описан Шантаи-Тотом и Середи (1982). Более современные и более сложные венгерские системы работают в настоящее время главным образом на машинах Siemens—7.755, IBM-3031 и VAX-11/80. К ним относится, в частности, описанная Бендлом и др. (1980) реализация МПролога, которая, как утверждается, в особенности приспособлена для модульного программирования. Вследствие быстроты, с которой создавались эти интерпретаторы для решения важных прикладных вычислительных задач, их авторы были вынуждены дополнить базисную марсельскую конструкцию целым рядом ad hoc механизмов, обеспечивающих, например, быстрое выполнение арифметических операций, оперирование файлами и интерфейс, предназначенный для использования программ, скомпилированных с языков низкого уровня. Эти средства реализовывались, по всей видимости, без особого учета семантической чистоты, в результате чего к уже имевшимся искажениям, вызываемым некоторыми из примитивов управления в марсельской конструкции, на основе которой первоначально строились венгерские системы, добавлялись новые возможные искажения. Некоторые проблемы, возникающие в связи с соединением Пролога с другими языками программирования, обсуждаются Середи (1981).

Важный вклад в технологию реализации был сделан в середине 70-х годов Браиохе (1976), который разработал методы сборки мусора для уплотнения стеков, а также реализовал в 1977 г. хвостовую рекурсию (вариант ОПВ, в котором определенные рекурсии заменяются итерациями). В дальнейшем ему удалось обойти проблемы повисания указателей при совместном использовании структур, и именно он написал первый интерпретатор без использования этого метода (Браиохе, 1982). Первая его система была написана на Паскале, а новая система, написанная на языке Си для операционной системы Юникс, реализована в 1979 г. Кроме того, Браиохе (1981) положил начало методам «интеллектуального возврата», которые основываются на проведении в период исполнения анализа зависимости вызовов, пред назначенного для обнаружения и устранения тупико-

вых вычислений. Более сложные схемы для выборочного возврата были впоследствии изобретены Перейрой и Порто (1982).

Тем временем важные успехи в несколько ином направлении были достигнуты за счет усилий Уоррена и его коллег из Эдинбургского университета. Марсельский Пролог функционировал там еще с 1974 г., и его недостатки побудили Уоррена разработать свой собственный компилятор с Пролога на машине DEC-10, получивший в настоящее время широкое и вполне заслуженное признание за искусную конструкцию и методологию программирования. К отличительным особенностям, впервые появившимся в этом компиляторе, относятся индексирование и компилирование входных процедур, двухстековая реализация совместного использования структур, а также большое число тщательно разработанных оптимизаций, направленных на сокращение как времени обработки, так и потребляемой памяти. Для предоставления пользователю двухцелевой среды, предназначенной и для разработки, и для исполнения его программ, средства реализации Пролога на DEC-10 включают сейчас как интерпретатор, так и компилятор. Обе эти компоненты написаны преимущественно на Прологе и приведены в действие посредством раскрутки. Конструкция первоначального компилятора описана Уорреном (1977а, 1977б). Последующие отчеты Уоррена и др. (1977, 1979) включают руководство для пользователей и сравнение свойств этой системы со свойствами типичных реализаций Лиспа. Средства оптимизации в DEC-10 Прологе описаны Уорреном (1979, 1980). В дальнейшем многие его разработки были перенесены на другие вычислительные устройства. Например, производные этой системы как с совместным использованием структур, так и без использования данного метода были установлены Мелишем (1982) в Эдинбургском университете на ЭВМ PDP-11, а многие из ее возможностей были включены в венгерскую систему MПролог.

Одним из самых быстрых среди всех имеющихся интерпретаторов является Ватерлоо-Пролог, созданный Робертсом (1977) в Университете Ватерлоо. Он работает на вычислительных машинах IBM-370/158, IBM-3031 и IBM-4341 и по своей структуре подобен марсельскому Прологу, однако технически более отшлифован. Впечатляющее быстродействие этого интерпретатора отчасти обусловлено тем, что он написан непосредственно на языке ассемблера, и отчасти — отсутствием в нем механизмов экономии пространства. Как и реализация Пролога на машине DEC-10, он ориентирован на конкретную архитектуру ЭВМ, что ограничивает его переносимость. Отчет Робертса о своей системе в особенности примечателен точностью, ясностью и полнотой, чего недостает почти всей остальной литературе, касающейся реализации логики как языка программирования.

Значительные продвижения в области реализации были сделаны также Кларком и Маккейбом из Имперского колледжа. Первым разработанным там интерпретатором был IC-Пролог. Написанный на Паскале и установленный на вычислительных машинах IBM-370 и CDC-6000, он использовался главным образом для проверки новых механизмов управления. В частности, с его помощью удалось подтвердить, что реализацию режимов управления, регулирующих сложные методы исполнения программ (например, параллельно и в режиме сопрограмм), можно осуществить без искажения логической семантики формальной системы. Описание IC-Пролога дается во многих статьях Кларка, Маккейба и др. (1979а, 1979б, 1980, 1982).

Последние работы Кларка и Маккейба велись в направлении реализации логики на микрокомпьютерах. Их интерпретатор микро-Пролога был первоначально создан для работы под управлением операционной системы CP/M на ЭВМ, собранных на процессоре Z80, таких как Research Machines 380Z и North Star Horizon. Интерпретатор написан на коде ассемблера Z80. В нем обеспечивается ОПВ и не применяется метод совместного использования структур. В настоящее время он адаптируется на широком классе других малых машин. Самоучители по микро-Прологу для начинающих были опубликованы Кларком и Маккейбом (1984), а также Энналсом (1984). В микро-Прологе используется логика хорновских дизъюнктов, усиленная за счет чистых расширений, которые позволяют достичь некоторой синтаксической общности полного языка логики предикатов первого порядка. Его стандартная версия будет работать с памятью емкостью в 64К байт, причем около 16К байт будет занимать интерпретатор ядра.

Японский проект создания ЭВМ пятого поколения добавил сильный импульс разработкам в области технологии реализаций для логического программирования. Поскольку этот проект оказывает такое значительное воздействие на всю область логического программирования в целом, мы обсудим его отдельно в следующей главе.

VIII. Вклад логического программирования в теорию вычислений

Вклад, который может сделать логическое программирование в общую теорию вычислений, не исчерпывается только еще одним языком программирования. Разумеется, логика применялась в теории вычислений и до появления логического программирования, но ее трудно было приспособить для использования на основном направлении развития этой теории, главным образом потому, что мы не имели тогда простой и реализуемой вычислительной интерпретации логики. Теперь ситуация полностью изменилась. Логику можно использовать для представления данных, программ, спецификаций и связей между ними; ее можно применять для описаний как на объектном уровне, так и на метауровне; она может использоваться для описания управления программным обеспечением, а также и для описания самого программного обеспечения. Значение логического программирования состоит в том, что оно позволяет автоматизировать все эти применения логики, сделать их основанными на общей вычислительной теории и, таким образом, внести единство в прежде разрозненные действия и инструментальные средства. В этой главе дается обзор некоторых приложений логического программирования к теории, практике и технологии вычислений.

VIII.1. Теория вычислений

Теория вычислений включает в себя ту часть математики, в которой рассматриваются эффективно вычислимые отношения. В ней изучаются математические свойства самих отношений, их определимость в той или иной формальной системе, вычислимость на конкретных машинах и их значения относительно конкретных схем интерпретации. Эти и другие аспекты данной теории традиционно развивались в рамках отдельных направлений, таких как теория рекурсивных функций, теория вычислимости, формальная семантика и т. д. Основной принцип, поддерживаемый многими сторонниками логического программирования, заключается в том, что оно позволяет объединить все эти различные подтеории, подвести под них общий фундамент

и дать им рационалистическое обоснование. Он поддерживается не только в том теоретическом смысле, что логика вообще лежит в основании математики, но и в более практическом смысле: логическое программирование, по-видимому, способно естественным образом охватить представительные и операционные характеристики широкого класса вычислительных формализмов и связанных с ними теорий. В этом разделе мы кратко очертили пути, на которых логическое программирование сделало вклад в теорию вычислений.

VIII.1.1. Представимость и вычислимость

В связи с каждым предлагаемым вычислительным формализмом естественно поставить вопрос: что с его помощью можно представлять и вычислять? Минимальным требованием почти для всех практических целей является возможность представления натуральных чисел. В логическом программировании это требование может быть удовлетворено наличием любой фиксированной константы, например константы 0 , для представления «нуля» и любого фиксированного функциона, например s , для представления «следующего числа». Натуральные числа можно тогда представить термами $0, s(0), s(s(0))$ и т. д. Менее тривиально, у нас может возникнуть также желание иметь возможность определять на любом таком представлении все те отношения, которые может потребоваться вычислять на множестве натуральных чисел. Возможность определения всех таких отношений была впервые доказана Робертом Хиллом. А именно он показал, что логики хорионских дизъюнктов достаточно для определения всех эффективно вычислимых функций на множестве $\{0, s(0), \dots\}$ и что для этой цели не требуется никаких других функционаров, кроме 0 и s . Более общее описание адекватности логики хорионских дизъюнктов дается в его отчете (Хилл, 1974).

Чаще, однако, при изучении вычислимых функций их определяют на областях, построенных из более богатого класса символов, чем $\{0, s\}$. Из любого конечного множества («алфавита»), содержащего константы, скажем $\{a, b\}$, и функционары, скажем $\{f, g\}$, можно построить область, собирая вместе все те термы, которые можно построить с помощью констант и функционаров из этого алфавита. Полученная таким образом область $\{a, b, f(a), g(a), f(b), f(g(a)), \dots\}$ называется эрбрановским универсумом. Аидрека и Немети (1976) показали, что каждую вычислимую на эрбрановском универсуме функцию можно определить посредством некоторой программы на языке логики хорионских дизъюнктов. Это обобщение результата Хилла было доказано с помощью того факта, что каждый эрбрановский универсум

является счетным, т. е. его можно поставить во взаимно однозначное соответствие с множеством натуральных чисел. Андрека и Немети показали далее, что логическую программу для вычисления функции над эрбрановским универсумом можно построить только с помощью тех констант и функторов, которые содержатся в алфавите этого универсума.

Наиболее известным понятием вычислимости является *тьюрингова вычислимость*: класс всех функций, которые могут быть «эффективно» (т. е. систематически и за конечное время) вычислены на любой машинной реализации любого формализма, отождествляется (посредством эмпирического предположения, известного как тезис Чёрча) с классом всех функций, вычислимых на универсальной машине Тьюринга (УМТ). Теория вычислимости логических программ в парадигме УМТ была впервые исследована Териундом (1977). Он построил около полуодюжины простых хорновских дизъюнктов, представляющих необходимый механизм УМТ, и доказал затем, что любая вычислимая по Тьюрингу функция может быть вычислена с помощью резолюции, исходя из этих дизъюнктов в ответ на соответствующие целевые утверждения. Кроме того, хорошо известный факт о том, что не существует никакого алгоритма, позволяющего решить, закончится или нет произвольное тьюрингово вычисление («проблема остановки»), преобразуется тогда в эквивалентный факт: не существует никакого алгоритма, позволяющего решить, дает ли произвольная программа на хорновских дизъюнктах конечное успешное резолютивное вычисление. Это в свою очередь просто вариант того факта, что общезначимость произвольных предложений в логике первого порядка не является полностью разрешимой.

Класс эффективно вычислимых функций можно также охарактеризовать как класс *частично рекурсивных функций*, которые строятся из некоторого множества базисных функций с помощью операций суперпозиции, примитивной рекурсии и минимизации. Себелик и Степанек (1980) показали, что все частично рекурсивные функции являются вычислимыми в логике хорновских дизъюнктов; они получили свое доказательство, просто представив базисные функции и правила построения частично рекурсивных функций в логике хорновских дизъюнктов. Они, а также Териунд показали, что вычислительная универсальность достижима даже в ограниченной форме логики хорновских дизъюнктов («бинарной» форме), в которой процедуры могут иметь не более одного вызова. Аналогичным образом ван Эмдеон (1977b) и Ковальским (1983b) были даны хорновские формулировки вычислимых функций, представимых с помощью рекурсивных равенств Клини и равенств Эрбрана — Геделя.

Одним из наиболее просто описываемых универсальных формализмов является класс алгоритмов Маркова. Каждый такой алгоритм многократно переписывает текущую строку символов x , получая всякий раз новую строку z путем замены некоторых вхождений подстроки u в x на другую строку v в соответствии с заданным множеством правил замены и продолжая делать эту операцию до тех пор, пока некоторая строка v не будет содержать специального символа окончания; данное множество правил характеризует функцию, которая по начальному состоянию строки x вычисляет заключительное состояние y . Легко представить эту схему в логике хорновских дизъюнктов с помощью дизъюнктов высокого уровня вида

переписать(x, y) если заменить(u, v), подставить(u, v, x, z),
продолжить(y, z, v)
продолжить(y, z, v) если указатель-коица(v) то $y = z$ иначе
переписать(z, y)

вместе с соответствующими дизъюнктами, описывающими предикаты подставить и указатель-конца. К этим дизъюнктам нужно только добавить тогда множество фактов заменить, таких как

заменить($A.x.B.NIL, C.x.NIL$)

характеризующих ту функцию, которую нужно вычислить. Оказывается, что стандартная стратегия исполнения логических программ полностью согласуется со стратегией выбора правил, обычно предписываемой алгоритмам Маркова. Исполнение начинается с постановки целевого утверждения ?переписать (x, y) с некоторой конкретной входной строкой на месте x . Читатель, имеющий склонность к теории, может восполнить детали этой формулировки (краткое и простое изложение алгоритмов Маркова дается Элсоном (1973)) и получить тем самым формальное доказательство ее эквивалентности алгоритмам Маркова и, следовательно, ее универсальности.

Все эти различные исследования ведут к убедительному (впрочем, неудивительному) подтверждению того, что логика хорновских дизъюнктов является универсальным вычислительным формализмом: по вычислительной силе она эквивалентна всем другим универсальным (и взаимно эквивалентным) формализмам, традиционно изучаемым в теории вычислимости. Вследствие этого, а также того, что логическое программирование завоевало сейчас прочные позиции как в теории, так и на практике, вовсе не обязательно — и, возможно, это не принесет особой пользы — применять для вычислительных целей нестандартные логики (такие как нечеткие, временные или модальные ло-

гики); более осмысленно попробовать сначала достичь выполнимой ими функции путем формулировки и реализации их в стандартной логике.

VIII.1.2. Семантика

Изучение семантики языка программирования в своей основе связано с приписыванием значений компонентам программ. В общем случае значение каждой такой компоненты определяется как ее контекстом в данной программе, так и некоторым фиксированным способом ее означивания с помощью этой программы. Например, из каждого множества Р логических процедур мы можем выбрать какой-либо предикативный символ, скажем p , и поставить вопрос, что «означает» символ p в соответствии с Р. Этот вопрос предполагает, что имеется соответствие, частично определяемое множеством процедур Р, между предикативными символами и некоторыми другими объектами, являющимися их значениями. Стало быть, значением p будет некоторый объект e . Мы назовем e денотатом p в Р: в соответствии с Р предикативный символ p обозначает (означает) e . Что же можно использовать в качестве таких объектов? Для вычислительной семантики очевидным выбором являются эффективно вычислимые отношения.

Первое всестороннее исследование семантики логических программ было предпринято ван Эмденом и Ковалльским (1976). Они разработали три различных вида семантики, в каждом из которых в качестве денотата предикативного символа определяется некоторое вычислимое отношение, и установили соотношения между ними. Семантика первого вида, называемая *операционной* (или *процедурной*) семантикой, задает в качестве денотата предикативного символа p в Р отношение, которому принадлежат все кортежи термов t , такие что предикат $p(t)$ выводим из Р с помощью какой-либо корректий системы вывода, т. е. отношение $\{t \mid P \vdash p(t)\}$. Выводимость $p(t)$ из процедур Р эквивалента существованию опровержения, демонстрирующего противоречивость программы $(P, G : ?p(t))$. Если рассматривать опровержение как вычисление и заметить, что для порождения этих вычислений имеются такие системы вывода, как резолюция, то термин «операционная семантика» становится оправданным. Это определение операционной семантики аналогично традиционному способу задания семантик обычных языков программирования — а именно, посредством определения механизма исполнения (как правило, на некоторой абстрактной машине) и объяснения затем значения программы, исходя из того, что вычисляется в результате применения к программе данного механизма. Короче говоря, значением предикативного символа p при таком подходе является отношение, вычисляемое программой

(P, G) с помощью корректного вывода. Наиболее привлекательная особенность этой семантики заключается в том, что несмотря на ее операционный характер, она не содержит тех сложностей, которые возникают при формулировке операционных свойств обычных процедурных языков, поскольку описание шага вывода намного проще описания механизмов управления в упомянутых языках.

Вторая семантика, определенная ван Эмдеоном и Ковальским, относится только к логическому программированию. В этой семантике деинтегром предикатного символа p в множестве процедур P является отношение, которому приадлежат все кортежи термов t , такие что $p(t)$ логически следует из P , т. е. отношение $\{t \mid P \vdash p(t)\}$. Заметим, что в этой семантике совершение не используются операционные понятия, и поэтому ее называют *непроцедурной семантикой*. В гл. I было показано, что понятие логического следствия базируется на понятиях интерпретации и модели. Сказать, что из P логически следует $p(t)$, — это значит сказать, что во всякой интерпретации (модели), в которой выполняются все процедуры из P , выполняется также и $p(t)$. Вследствие этого вторую семантику называют *теоретико-модельной семантикой*. Больше всего она привлекательна своей простотой: значением логической программы является то, что логически следует из ее утверждений.

Третья семантика также непроцедурная, но она полностью аналогична семантике, обычно задаваемой для программ, написанных на языке определений рекурсивных функций. Ее формализация основывается на несколько более сложных понятиях, чем те, которые потребовались для определения двух других семантик. Эти понятия, по-видимому, лучше всего ввести в связи с простым (хотя и довольно искусственным) примером. Рассмотрим следующую процедуру C , выбранную из некоторого множества процедур P :

$C : p(x, y) \text{ если } q(x), r(y)$

Пример C' процедуры C получается в результате одновременной подстановки каких-либо термов вместо переменных из C . Так, пример

$C' : p(x, B) \text{ если } q(x), r(B)$

получается из C подстановкой B вместо y . Основным примером C называется такой пример C , который не содержит переменных, так что каждый атом (предикат) в нем является основным атомом. Например:

$C' : p(A, B) \text{ если } q(A), r(B)$

Рассмотрим теперь множество P^* всех основных примеров всех процедур из P , которые можно получить, подставляя вместо их

переменных термы из эрбрановского универсума $H(P)$, построенного с помощью констант и функций, входящих в P . Так, например, если A и B — единственные константы, встречающиеся в P , и P не содержит функций, то $H(P)$ — это множество термов $\{A, B\}$, а основными примерами процедуры C , которые входят в множество P^* , являются

- $p(A, A)$ если $q(A), r(A)$
- $p(A, B)$ если $q(A), r(B)$
- $p(B, A)$ если $q(B), r(A)$
- $p(B, B)$ если $q(B), r(B)$

Таким способом мы готовим интерпретацию множества процедур P над областью $H(P)$ подобно тому, как это описано в гл. I. Каждому атому, являющемуся антецедентом некоторой процедуры из P^* , произвольным образом сопоставим истинностное значение t или f ; единственное ограничение здесь, разумеется, состоит в том, что одинаковым атомам должно быть сопоставлено одно и то же истинностное значение. Далее, если всем атому, являющемуся антецедентами некоторого примера из P^* , сопоставлено значение t , то и атому из консеквента также сопоставим значение t . Наконец, всем тем атому, которые являются консеквентами процедур из P^* и которым не было сопоставлено значение t , сопоставим значение f . Обозначим через I множество всех атомов-антецедентов, которым сопоставлено значение t , а через J — множество всех атомов-консеквентов, которым также сопоставлено t . Можно показать тогда, что интерпретация P , основанная на таком распределении истинностных значений, является моделью P в том и только том случае, когда $J \subseteq I$ ¹⁾.

Интерпретация, получаемая сопоставлением истинностных значений тем примерам атомов из P , которые образуются, как и выше, путем подстановки вместо их переменных термов из $H(P)$, называется *эрбрановской интерпретацией*. (В технической литературе эрбрановскую интерпретацию зачастую определяют как произвольное множество основных атомов, образованных путем подстановки термов из $H(P)$ вместо переменных

¹⁾ Здесь автор не совсем точен. Я приведу все необходимые формулировки, следуя ван Эмдену и Ковальскому (1976). Пусть I — произвольное множество основных атомов, полученных подстановками термов из $H(P)$ вместо переменных в атомах из P . Множество I можно рассматривать как (эрбрановскую) интерпретацию P над областью $H(P)$: основной атом считается истинным (принимает значение t), если он принадлежит I , и ложным (принимает значение f), если он не принадлежит I . Определим J как множество всех тех основных атомов A , для которых имеется процедура A если B_1, \dots, B_n из P^* , такая что $B_1 \in I, \dots, B_n \in I$ ($n \geq 0$). В частности, J содержит все основные примеры фактов из P . Нетрудно доказать, что I — эрбрановская модель P тогда и только тогда, когда $J \subseteq I$. — Прим. перев.

атомов из P ; так можно коротко говорить о тех основных атомах, которым присвоено значение t .) Эрбрановская интерпретация, в которой выполняются все процедуры из P , называется тогда *эрбрановской моделью* P .

При таком подходе как только выбрано множество I , множество J становится однозначно определенным данными выше правилами. Другими словами, существует функция T , отображающая множества атомов в множества атомов, такая что $T(I) = J$. Этую функцию обычно называют *преобразованием*, соответствующим P . Мы приходим теперь к важной теореме: для каждого множества процедур P существуют множества атомов I , удовлетворяющие условию $T(I) = I$ и называемые *неподвижными точками* преобразования T ; среди всех этих множеств существует ровно одно множество, которое является подмножеством всех остальных; его называют *наименьшей неподвижной точкой* (*least fixpoint*) преобразования T и обозначают посредством $\text{Ifp}(T)$. Итак, произвольное множество процедур P имеет много моделей, включая эрбрановские модели I , каждая из которых удовлетворяет условию $T(I) \subseteq I$. Наименьшая модель I , называемая наименьшей эрбрановской моделью, оказывается также наименьшей моделью, удовлетворяющей условию $T(I) = I$. Она является поэтому наименьшей неподвижной точкой преобразования T .

Денотат каждого предикатного символа p из P можно определить теперь как отношение $\{t | p(t) \in \text{Ifp}(T)\}$. Это определение, введенное ван Эмденом и Ковальским, характеризует *семантику неподвижной точки* логических программ. В этой семантике множество процедур P рассматривается в терминах соответствующего равенства неподвижной точки $I = T(I)$. Его можно читать следующим образом: кортежи термов, вычисляемые с помощью P (проще говоря, решения атомов-консеквентов из процедур, входящих в P), суть в точности те кортежи, которые могут быть вычислены с помощью вызовов (атомов-антecedентов) из P ; совокупным значением всех предикатных символов из P считается множество отношений, образованных из этих кортежей, которое составляет наименьшее множество I , удовлетворяющее равенству неподвижной точки. Такое прочтение полностью аналогично интерпретации неподвижной точки программ вида $f(z) := T[f](z)$ в языке определений рекурсивных функций: значением функции f в программе считается та единственная функция, которая является наименьшей неподвижной точкой преобразования T .

Для того чтобы лучше понять сущность преобразования T , полезно рассмотреть следующий способ построения совокупности основных фактов (атомов). Пусть вначале эта совокупность является пустым множеством \emptyset . Тогда $T(\emptyset)$ добавит к нашей

совокупности все основные примеры фактов, входящих в Р. Еще одно применение преобразования Т дает множество $T(T(\emptyset)) = T^2(\emptyset)$, которое включает в себя все основные факты, выводимые путем однократного применения резолюции снизу вверх к процедурам из Р и фактам из $T(\emptyset)$. В общем случае каждое последующее применение преобразования Т дает те факты, которые являются непосредственными логическими следствиями процедур из Р и уже полученных фактов. По этой причине преобразование Т стали называть *функцией непосредственного следования*, соответствующей Р. Множество всех основных атомов, являющихся логическими следствиями Р, представимо тогда в виде объединения всех множеств $T^i(\emptyset)$, где i пробегает множество натуральных чисел. Эти следствия, разумеется, суть в точности те вызовы, которые решаются с помощью Р; объединение множеств $T^i(\emptyset)$ охватывает, следовательно, всю совокупность кортежей термов, вычисляемых исходя из Р. Так как мы уже потребовали, чтобы этим множеством было $\{t \mid t \in R\}$, то отсюда вытекает, что мы должны иметь $Ifp(T) = \bigcup_i T^i(\emptyset)$. Это в точности соответствует хорошо известной характеристизации наименьших неподвижных точек, данной Клини (1952).

Основное достижение исследования ван Эмдена и Ковальского состоит в получении элегантных доказательств эквивалентности операционной семантики, теоретико-модельной семантики и семантики неподвижной точки для логических программ на языке хорновских дизъюнктов. Эти семантики эквивалентны в том смысле, что они определяют одни и те же денотаты для предикатных символов. Эквивалентность первых двух семантик получается благодаря теореме Гёделя о полноте логики предикатов первого порядка¹⁾, связывающей доказуемость с общезначимостью. Эквивалентность второй и третьей семантик устанавливается посредством доказательства того, что $Ifp(T)$ есть наименьшая эрбрановская модель Р и что $R \models p(t)$ тогда и только тогда, когда предикат $p(t)$ является истинным в этой модели.

В последующей статье Айт и ван Эмден (1982) представили семантику неподвижной точки на формальной основе теории полных решеток и с помощью этого доказали затем корректность и полноту стандартной стратегии исполнения для программ в языке хорновских дизъюнктов. Они определили также *наибольшую неподвижную точку* преобразования Т и использовали ее для характеристики неразрешимых программ, которые за конечное время приходят к неудаче из-за отсутствия успешных вычислений в пространстве поиска.

¹⁾ Согласно этой теореме существуют такие (корректные и полные) системы вывода, что $R \models p(t)$ тогда и только тогда, когда $R \vdash p(t)$. — Прим. перев.

Систематические и формальные изложения применений теории неподвижной точки логических программ можно найти в монографиях Кларка (1979) и Ллойда (1984). Семантический анализ логических программ на основе *оптимальных неподвижных точек* недавно был дан Лассезом и Маером (1983).

VIII.1.3. Корректность и полнота стратегии исполнения

Главная привлекательность логики как языка программирования с точки зрения технологии реализации состоит в том, что стратегии исполнения программ могут быть строго проанализированы в хорошо разработанных рамках теории доказательств. В частности, интерпретаторы, имеющие целью реализацию различных вариантов стандартной стратегии, можно проверить с помощью известных результатов теории систем резолютивного вывода. Далее мы приведем некоторые результаты о корректности и полноте, установленные для резолюции в качестве стратегии исполнения.

В своей наиболее общей форме резолюция применяется к произвольным дизъюнктам вида $L_1 \vee L_2 \vee \dots \vee L_m$, где L_i — это либо положительная литер (атом, предикат), либо отрицательная литер (атом, стоящий под отрицанием). На каждом шаге берутся два дизъюнкта (родительские дизъюнкты), такие что некоторая положительная литер в одном из них и некоторая отрицательная литер в другом содержат предикаты, унифицируемые некоторой подстановкой Θ ; мы говорим, что данные литеры «отрезаются». В результате этого шага получается третий дизъюнкт (резольвента) $C\Theta$, где C — дизъюнкция всех литер (если, конечно, они имеются) из родительских дизъюнктов, отличных от тех, которые отрезаются. Выбор родительских дизъюнктов и отрезаемых литер осуществляется произвольным образом. Как показал Робинсон (1965), метод резолюций является корректным и полным в том смысле, что пустой дизъюнкт \Box выводим из входного множества дизъюнктов S посредством некоторой конечной последовательности шагов резолюции тогда и только тогда, когда множество S невыполнимо.

В большинстве современных систем логического программирования входное множество S ограничивается только хориовскими дизъюнктами, каждый из которых содержит не более одной положительной литеры. На каждый шаг резолюции также накладываются ограничения: (i) один из родительских дизъюнктов должен быть фактом или импликацией (т. е. некоторой процедурой), (ii) другой родительский дизъюнкт должен быть самой последней полученной резольвентой и текущим целевым утверждением и (iii) должно иметься некоторое фиксированное правило выбора, с помощью которого единственным образом

определяется отрезаемый вызов во втором родительском дизъюнкте. Так как в (i) в качестве родительских дизъюнктов выбираются так называемые определенные (Definite) дизъюнкты (имеющие ровно одну положительную литеру), а условиями (ii) и (iii) характеризуется SLD-результатия — линейная (Linear) резолюция с функцией выбора (Selector function), то эту систему вывода называют *SLD-результатией* (прежде ее называли *LUSH-результатией*). Впервые она была описана Ковалевским (1974b); более подробно различные варианты SLD-результатации представлены в статье Ковалевского и Кунера (1971). Корректность и полнота общей результатации позволяют в качестве частного случая довольно просто доказать, что SLD-результатия также является корректной и полной в том простом смысле, что пустой дизъюнкт \square выводим из множества дизъюнктов S тогда и только тогда, когда S невыполнимо.

Более детальная характеристика корректности и полноты SLD-результатации получается путем включения в рассмотрение как решений целевого утверждения (подстановок, дающих ответ), вычисляемых посредством вывода пустого дизъюнкта \square , так и правила вычислений, используемого для их нахождения. Прежде всего, определим дающую правильный ответ подстановку Θ для программы $(P, G : ?g_1, \dots, g_n)$ как такую подстановку Θ , для которой из множества процедур P логически следует универсальное замыкание формулы $(g_1, \dots, g_n)\Theta$. Если, к примеру, G есть целевое утверждение $?g(x)$ и из P логически следует $(\forall y)g(t(2, y))$, то $\Theta = \{x := t(2, y)\}$ является подстановкой, дающей правильный ответ для программы (P, G) . Правило вычислений — это любое фиксированное правило, однозначно определяющее, какие из вызовов, содержащихся в текущем целевом утверждении, отрезаются на каждом шаге SLD-результатации. Например, стандартным правилом вычислений в Прологе является правило «выбрать первый вызов». Следующий результат о (слабой) полноте SLD-результатации был установлен Хиллом (1974).

Каждая подстановка Θ , дающая правильный ответ для логической программы, вычисляется SLD-результатией с помощью некоторого правила вычислений.

Заметим, что в этой теореме не утверждается, что существует хотя бы одно правило вычислений, которое позволяет получить подстановки, дающие все правильные ответы для программы; в ней говорится только то, что множество всех возможных правил вычисления покрывает все правильные подстановки Θ .

На самом деле существует более сильный результат о (сильной) полноте SLD-результатации, который устанавливает, что множество вычисляемых подстановок, дающих правильные ответы,

не зависит от используемого правила вычисления: этот результат особенно полезен, поскольку он гарантирует полноту сложных стратегий выбора вызова, таких как исполнение в сопрограммном режиме под управлением потока данных. Из этого результата и определения корректности резолюции вытекает теорема, гарантирующая корректность и полноту SLD-резолюции:

Какое бы правило вычислений ни применялось к программе, Θ является подстановкой, дающей правильный ответ для программы, тогда и только тогда, когда она вычислима посредством вывода пустого дизъюнкта \square с помощью SLD-резолюции.

Различные доказательства этой, а также связанных с ней теорем были даны Аптом и ван Эмдеоном (1982), Кларком (1979) и Ллойдом (1984). Особый интерес представляет использование в этих доказательствах семантики неподвижной точки и теоретико-модельных свойств логических программ; теория верификации для стратегии исполнения черпает силу непосредственно из строго установленных и взаимосвязанных характеризаций семантик языка логического программирования.

Следует отметить, что эти различные результаты о корректности и полноте применимы только к системе SLD-вывода, а не к какой-либо конкретной процедуре доказательства, построенной на ее основе путем наложения условий на стратегию поиска. Как мы уже видели в гл. V, стандартная процедура доказательства в Прологе является *несправедливой*, поскольку используемое в ней правило поиска в глубину с возвратом при наличии бесконечного пространства поиска не обеспечивает полноту лежащей в основе системы SLD-вывода, хотя и гарантирует ее корректность. Практическое значение этого факта состоит в том, что для верификации каждого логического интерпретатора требуется анализ его правила поиска, а также уверенность в стандартных результатах о корректности и полноте резолюций. Примером реализации справедливой стратегии исполнения посредством применения поиска в ширину является система Логлисп, описанная Робинсоном и Сибертом (1980), в которой логика погружается в Лисп.

VIII.1.4. Отрицание

В гл. III мы отмечали, что при некоторых ограничениях логический интерпретатор может без риска исполнять квазитрицательные (\sim) вызовы процедур, интерпретируя неудачу как отрицание: $\neg p$ выводится из неудачной попытки вывести p . Этот механизм операционно очень эффективен. Он избавляет программиста от утомительной работы, связанной с явным

представлением «отрицательной» информации в своих программах. Наоборот, все то, о чем он не сказал в своей программе, считается логически ложным. Достигаемая экономия в стиле программирования оказывается, несомненно, существенной в таких приложениях, как базы данных. Мы хотим указать, что входит в базу данных, но не желаем перечислять бесконечную совокупность вещей, которые в ней не содержатся.

Тем не менее дедуктивная сила, получаемая в результате использования правила *отрицание как неудача*, существенно меньше той, которую дает строгое классическое отрицание (\neg), и поэтому значительные усилия были направлены на изучение проблемы точного определения различий между \sim и \neg . Заметим, что дизъюнкт общего вида, являющийся произвольной дизъюнкцией литер, можно представить в следующих эквивалентных формах:

$$\begin{aligned} A_1 \vee \dots \vee A_m \vee \neg B_1 \vee \dots \vee \neg B_n \\ A_1 \vee \dots \vee A_m \text{ если } B_1, \dots, B_n \\ A \text{ если } B_1, \dots, B_n, \neg A_2, \dots, \neg A_m \end{aligned}$$

Ограничение, налагаемое на хорновские дизъюнкты ($m \leq 1$), эквивалентно запрету на использование отрицательных вызовов $\neg A_i$ в третьей из приведенных выше форм. Можно было бы подумать, что поскольку дизъюнкты общего вида по своей выразительной силе эквивалентны стандартной логике предикатов первого порядка, данное ограничение должно было бы повлечь за собой некоторую потерю выразительной силы по сравнению с логикой первого порядка. Тем не менее этого, очевидно, не может случиться, так как мы знаем из разд. VIII. 1.1, что в логике хорновских дизъюнктов определяются все вычислимые отношения. Что мы вместо этого теряем, так это некоторую свободу стиля. Отрицание как неудача позволяет частично компенсировать указанную потерю. Это один из вариантов решения так называемой «проблемы отрицания» в логическом программировании, состоящей в нахождении таких расширений логики хорновских дизъюнктов, которые (i) позволяют использовать формулы, близкие к дизъюнктам общего вида, (ii) могут быть эффективно реализованы, которые (iii) могут быть совмещены со стандартной процедурной интерпретацией, и (iv) соответствуют логической семантике (и, следовательно, не нарушают требования корректности и полноты).

Законность использования отрицания как неудачи была впервые установлена Кларком (1978). Он доказал, что это правило сохраняет корректность, если принимается *допущение замкнутости мира* (ДЗМ), согласно которому множество процедур Р в программе представляет все имеющиеся знания об упоминаемых в ней отношениях. Это допущение эквивалентно

предположению о том, что множество P неявно расширено за счет так называемого *замыкания* (*completion*) P , которое обозначается через $\text{comp}(P)$ и состоит из аксиоматизации отношения тождества (\equiv) и всех предложений, получаемых из процедур P заменой связки *если* на «только если». Например, если P есть множество

$$\begin{aligned} P1 &: a(y) \text{ если } b(y) \\ P2 &: a(2) \\ P3 &: b(3) \end{aligned}$$

то его замыкание $\text{comp}(P)$ содержит предложения

$$\begin{aligned} C1 &: (b(y) \vee y=2) \text{ если } a(y) \\ C2 &: y=3 \text{ если } b(y) \end{aligned}$$

так что из объединения P и $\text{comp}(P)$ вытекают следующие определения отношений *a* и *b*:

$$\begin{aligned} D1 &: a(x) \Leftrightarrow (\exists y)(x=y, b(y)) \vee x=2 \\ D2 &: b(x) \Leftrightarrow x=3 \end{aligned}$$

Заметим, что из этого расширенного множества логически следуют отрицательные факты, такие, как $\neg b(1)$, не являющиеся строгими следствиями одного лишь множества P . Кларк обосновал использование правила отрицание как неудача, опираясь на точку зрения, согласно которой программист имеет в виду, что его программа утверждает D1—D2, но пишет на самом деле только процедуры P1—P3 и опускает предложения C1—C2, поскольку для вычислений они не нужны.

Чтобы привести результаты о корректности и полноте для этого правила, нам потребуются два предварительных определения. Эрбановский базис $B(P)$ множества процедур P — это множество всех основных предикатов (атомов), получающихся подстановками термов из эрбановского универсума $H(P)$ вместо переменных в предикатах из P ; по существу эрбановский базис охватывает все основные вызовы, которые предположительно можно было бы пожелать исследовать с помощью P . *Множество финитно-неудачных вызовов* (*finite failure set*) $F(P)$ множества процедур P — это множество всех тех основных предикатов из $B(P)$, которые, будучи поставлены в качестве целевых утверждений и исполнены вместе с процедурами P при помощи SLD-резолюции, приводят к неудаче (т. е. являются неразрешимыми) за конечное число шагов, или, иначе, дают конечные пространства поиска, не содержащие опровержений. Правило отрицание как неудача является тогда корректным, поскольку для стандартного множества хорновских процедур P

мы имеем

$$\text{comp}(P) \models \neg A \text{ если } A \in F(P)$$

Это значит, что если основное целевое утверждение $? \sim A$ исполняется стандартным образом с помощью попытки решить вызов A и если эта попытка за конечное число шагов приводит к неудаче, то из допущения замкнутости мира корректно будет вывести $\neg A$. (На самом деле доказательство Кларка является более общим — оно позволяет квазиотрицательным вызовам входить не только в исходное целевое утверждение, но и в тела дизьюнктов из P .) Это происходит потому, что $\text{comp}(P)$ — неявная часть исполняемой программы — логически влечет тогда $\neg A$.

Данный анализ был ограничен только основными вызовами, но это ограничение не является критическим. Если целевое утверждение $? \sim A$ содержит переменные, то неудача (за конечное число шагов) при попытке решить последующий вызов A без каких-либо трудностей решает это целевое утверждение. Если же, однако, вызов A был решен, причем хотя бы одной из его переменных было присвоено какое-то значение, то (как объяснялось в гл. III) следовало бы выдать сообщение об ошибке управления — этот результат потенциально приводит к потере полноты, хотя корректность здесь не теряется.

Свойства полноты правила отрицание как неудача оказались намного более проблематичными. Джффар, Лассез и Ллойд (1983) представили недавно довольно сложное доказательство полноты правила отрицание как неудача для стандартного множества процедур P , которая заключается в том, что

$$A \in F(P) \text{ если } \text{comp}(P) \models \neg A, A \in B(P)$$

т. е. основное целевое утверждение $? \sim A$ с необходимостью решается посредством неудачного решения (за конечное число шагов) вызова A всякий раз, когда из допущения замкнутости мира следует $\neg A$. Возможную потерю полноты в том случае, когда $\sim A$ содержит переменные, мы уже отмечали. Еще хуже то, что полнота подвергается опасности, когда квазиотрицательные вызовы встречаются в телях дизьюнктов из P . Это происходит потому, что при такой операционной трактовке связки \sim в правиле отрицание как неудача формулы A , $\sim \sim A$ и $A \text{ если } \sim A$ не являются эквивалентными, как это имеет место, когда связка \sim классически интерпретируется как \top . Вследствие этого вызов $\sim A$, который согласно ДЗМ должен был бы решаться, может на самом деле приводить к незавершающемуся исполнению, и в этом случае мы не будем иметь $A \in F(P)$. Отсюда ясно видно, что хотя отрицание как неудача и оправдывается использованием ДЗМ, фактически оно слабее этого

допущения.. Превосходное изложение обсуждаемой проблематики дается Ллойдом (1984), тогда как примеры простых патологических программ, которые выявляют проблемы, возникающие из-за разрешения использовать связку \sim в телах дизъюнктиов, можно найти в книге Ковальского (1979а).

Другой очень интересный подход к проблеме отрицания был недавно предложен Габбаем и Серготом (1984). Они указывают, что правило отрицание как неудача ограничено либо подтверждением, либо отрицанием каждого конкретного вызова и не способно обслужить третью возможность — сделать вывод о том, что вызов не является ни подтверждаемым, ни отрицаемым. Они предлагают новую форму отрицания, обозначаемую здесь посредством \neg^* , которая имеет значение «ведет к нежелательным последствиям (таким как противоречивость)». База данных из фактов и правил, над которой исполняется целевое утверждение, состоит в их подходе из двух частей Р и N: в Р содержатся положительные (истинные) факты, тогда как в N содержатся отрицательные (ложные) факты. Значение связки \neg^* можно определить теперь следующим образом:

$$(P, N) \vdash \neg^* A \Leftrightarrow (P, A) \vdash B, \text{ где } B \in N$$

Таким образом, отрицание A нельзя вывести просто из невозможности показать A с помощью (P, N); вместо этого отрицание A можно вывести, только показав, что допущение (истинности) A ведет вместе с P к некоторому отрицательному факту из N.

Эта схема названа авторами «отрицание как противоречие», поскольку заключение $\neg^* A$ выводится на основе доказательства того, что A несовместимо с P, т. е. из допущения истинности A и P логически следует некоторое утверждение B, которое программист явно классифицировал как ложное, поместив в N. Габбай и Сергот показали, что отрицание как неудача и отрицание как противоречие совпадают, когда N ограничено только теми предикатами A, которые принадлежат множеству финитно-недущих вызовов для процедур P. Они доказали также, что отрицание как противоречие слабее классического отрицания. А именно, они показали, что все еще можно построить множества P и N, для которых $(P, N) \vdash A$, однако вызов $\neg^* \neg^* A$ за конечное число шагов приводит к неудаче, если исходить из (P, N) .

VIII.1.5. Рассуждения на метауровне

Одно из наиболее интересных свойств логики, имеющее потенциально далеко идущие следствия, — это возможность представлять в ней понятия *метауровня*. Простым примером такого понятия в контексте логического программирования могло бы

быть понятие вычислимости (выводимости) из множества процедур. Рассмотрим, к примеру, следующее предложение, описывающее правило отрицание как неудача:

из y выводится $\neg x$ если из $y \neg$ выводится x

Здесь y , x и $\neg x$ обозначают элементы системы объектного уровня, в данном случае — различные утверждения программы на *объектном языке* (ОЯ) хорновских дизьюнктов. С другой стороны, символы **выводится**, **не выводится** и **если** являются элементами *метаязыка* (МЯ), на котором мы описываем свойства системы объектного уровня.

Если мы рассматриваем ОЯ как язык, в котором составляются и исполняются конкретные программы, то МЯ становится языком для рассуждений о составлении и исполнении программ, т. е. МЯ становится языком, в котором мы можем строить средства манипулирования программами, такие как интерпретаторы, верификаторы, редакторы или операционные системы. Метаязык, несомненно, может многое дать для формализации и реализации таких задач, как построение программ, их анализ и исполнение, — короче говоря, для технологии программного обеспечения. Особенно интересным является тот факт, что метаязык может сам быть просто языком логики хорновских дизьюнктов, и, таким образом, его можно реализовать при помощи уже имеющихся логических реализаций. Эта мощная двойственная роль логики была обнаружена и использована уже на раннем этапе развития логического программирования: простые интерпретаторы, написанные на обычных языках программирования, применялись для того, чтобы посредством раскрутки получить более совершенные версии, которые сами были написаны (что более удобно) на языке логики и которые содержали метауровневые описания того, как должны исполняться другие логические программы.

Все существующие на сегодняшний день интерпретаторы предлагают программисту некоторые возможности для написания логических программ, манипулирующих другими программами. Эти возможности основаны на простом приеме — использовании термов определенного вида для представления компонент программ. Например, структурированный терм $p(x)$, входящий в программу метауровня, можно использовать для представления предиката $p(x)$ из некоторой программы объектного уровня. Целый ряд мощных преобразований программ, осуществляемых посредством передачи таких термов через метапеременные, объясняется Кларком и Маккейбом (1984) для системы микро-Пролога, философия проектирования которой в значительной степени базируется на возможностях метауровня.

Применение единственного языка, играющего роль как ОЯ, так и МЯ, называется *амальгамированием*, и, для того чтобы оно было эффективным, следует воспользоваться некоторым определением Pr (записанным в этом языке) *отношения доказуемости*. Впервые это определение Pr для логики хорновских дизъюнктов было построено и исследовано Ковалевским (1979а). В основе его конструкции при самой простой формулировке доказуемости лежит предикат **demo**(«*x*», «*y*»), который читается как

предложение *y*, называемое «*y*», доказуемо
из предложения *x*, называемого «*x*».

Pr и предназначено для того, чтобы дать определение отношения **demo**. Условие для достижения этой цели можно представить следующим образом:

$$\text{Pr} \vdash \text{demo}(\langle x \rangle, \langle y \rangle) \Leftrightarrow x \vdash y$$

Ковалевский отождествил это условие с *принципом рефлексии*, предложенным Вейраухом (1980). Выражаемое им требование состоит в том, чтобы *имитация* на метауровне (доказательства *y* с помощью *x*), которая является результатом исполнения программы ($\text{Pr}, ?\text{demo}(\langle x \rangle, \langle y \rangle)$) на метауровне, была эквивалентна непосредственному исполнению программы ($x, ?y$) на объектном уровне. Используя логику хорновских дизъюнктов, утверждение метауровня, выражающее правило вывода отрицание как неудача, можно было бы записать тогда следующим образом

$$\text{demo}("x", "ne(y)") \text{ если } \sim \text{demo}(x, y)$$

где «*ne* (*y*)» и «*x*» можно было бы заменить на некоторые подходящие конкретные термы, представляющие отрицаемый (\neg) предикат и множество процедур соответственно. Такого рода утверждение могло бы затем стать частью логического интерпретатора, который сам написан на языке логики и определяет стратегию исполнения программ объектного уровня с помощью отношения **demo**.

Более современное описание металогических возможностей логики хорновских дизъюнктов и их приложений к рассуждениям о программах дается Боузном и Ковалевским (1982). Они обсуждают также возможности использования ОЯ-МЯ амальгамирования для разрешения некоторых проблем, связанных с давней критикой классической логики, а именно, с ее немонотонностью (см., например, Минский, 1975; Бобров, 1980) и ее восприимчивостью к проблеме фреймов (см., например, Рафаэл, 1971). Мы рассмотрим кратко обе эти проблемы ниже.

Классическая логика является *монотонной* в том смысле, что в результате добавления к произвольному множеству пред-

ложений S некоторого другого предложения s множество доказуемых из S следствий никогда не уменьшается, а, наоборот, скорее увеличивается. Мы можем выразить этот факт символически следующим образом:

$$\text{для всех } S, s \text{ и } c \quad S \cup \{s\} \vdash c \text{ если } S \vdash c$$

откуда становится очевидным, что монотонность является свойством логического отношения доказуемости \vdash . Основные возражения против свойства монотонности возникают обычно в связи с приложениями, включающими динамически меняющиеся базы знаний. В качестве тривиального примера допустим, что текущее состояние базы знаний S позволяет нам вывести некоторое заключение c . В зависимости от используемой системы вывода c может оказаться необходимым логическим следствием S , однако может быть и так, что за неимением противоположных знаний c выводится просто как правдоподобное следствие. Пусть теперь систему явным образом информировали о том, что имеет место $\neg c$, добавив, например, предложение $s = \neg c$ к множеству S . В этом новом состоянии системы мы могли потребовать, чтобы c было теперь невыводимым, т. е. потребовать немонотонного поведения. Но классическая логика так себя не ведет, поскольку из ее монотонности вытекает, что предложение c должно оставаться выводимым из $S \cup \{\neg c\}$. На самом деле, если имеет место $S \vdash c$, то расширенная база знаний $S \cup \{\neg c\}$ обязательно будет противоречивой. Ковальский (1979а) утверждает, что наивное возражение против свойства монотонности классической логики можно было бы преодолеть, рассматривая противоречивость как естественный и полезный результат определенного рода переходов в эволюции базы знаний и применяя ее позитивно для управления последующим восстановлением непротиворечивости, например, путем определения допущений, которые следовало бы теперь отбросить, изменить или как-то иначе понизить в статусе. В статье Боэна и Ковальского (1982) дается интересная иллюстрация этого тезиса на примере использования ОЯ-МЯ амальгамирования в контексте простой системы управления базами данных. К сожалению, эффекты немонотонности склонили других исследователей обратиться к иным (обычно многозначным) формам логики или даже полностью отказаться от нее.

Некоторое смущение может вызвать тот факт, что существующие интерпретаторы не могут, во всяком случае, вести себя монотонно. Правило отрицание как неудача, например, не является монотонным, поскольку всегда имеется возможность так расширить множество предложений, чтобы некоторое прежде невыводимое предложение c стало выводимым, и в этом случае прежнее следствие $\sim c$ больше таковым являться не будет.

Поэтому на практике логическое программирование критиковалось как за то, что оно монотонно, так и за его немонотонность в зависимости от точки зрения критика.

Проблему фреймов в логике также можно рассматривать как проблему, связанную с представлением знаний и внесением изменений в базы знаний. Чтобы пояснить это, достаточно будет простого примера. Пусть мы желаем, чтобы в базе знаний был представлен динамически меняющийся список, который в некоторый момент времени имел бы вид $L = (A, B, C)$ и представлялся подобно массиву с помощью следующего множества фактов

$$\begin{aligned} &\text{элемент}(A, 1) \\ &\text{элемент}(B, 2) \\ &\text{элемент}(C, 3) \quad \text{длина}(3) \end{aligned}$$

Допустим далее, что мы захотели расширить этот список, добавив к нему четвертый элемент D . Стало быть, следующее его состояние должно представляться множеством фактов

$$\begin{aligned} &\text{элемент}(A, 1) \\ &\text{элемент}(B, 2) \\ &\text{элемент}(C, 3) \\ &\text{элемент}(D, 4) \quad \text{длина}(4) \end{aligned}$$

Проблема фреймов возникает в связи с попыткой описать этот процесс расширения списка на языке логики, поскольку очевидно, что для этого необходимо каким-то образом связать старое и новое состояния базы знаний и дать им имена. Поэтому вместо того, чтобы писать, как обычно, $\text{элемент}(u, i)$, мы пишем предикат $\text{элемент}(u, i, L)$, означающий, что L — это состояние, в котором u является i -м элементом. Вместо предиката $\text{длина}(n)$ мы также пишем предикат $\text{длина}(L, n)$, означающий, что длина списка в состоянии L равна n . Если обозначить состояние, которое получается в результате добавления элемента v к L , посредством $\text{добр}(L, v)$, то это новое состояние можно соотнести со старым при помощи таких предложений

$$\begin{aligned} &\text{элемент}(u, i, \text{добр}(L, v)) \quad \text{если } \text{элемент}(u, i, L) \\ &\text{элемент}(v, i, \text{добр}(L, v)) \quad \text{если } \text{длина}(L, i - 1) \\ &\text{длина}(\text{добр}(L, v), i) \quad \text{если } \text{длина}(L, i - 1) \end{aligned}$$

Первое из этих предложений является примером *аксиомы фрейма*: она служит для того, чтобы определить те факты в базе знаний, которые сохраняются (переиносятся) при переходе к новому состоянию. Для формулировок задач, связанных с большим пространством состояний, может потребоваться много различных классов фактов, которые должны сохраняться таким способом. В рассматриваемом примере мы объявляем, что если u — i -й

элемент в одном состоянии, то он продолжает оставаться i -м элементом всякий раз, когда в результате добавления еще одного элемента список переходит в новое состояние.

Ковалевский (1979а) выделяет два аспекта проблемы фреймов: во-первых, многочисленность аксиом фреймов, требуемых для реальных приложений, и, во-вторых, трудность эффективного управления ими, когда они вызываются «снизу вверх» при проведении прямого рассуждения от исходного состояния к целевому. Предлагаемые им средства для преодоления указанных трудностей заключаются соответственно в том, чтобы использовать единственную обобщенную аксиому фрейма, имеющую возможность доступа к отдельной базе данных, где представлены все те свойства, которые должны сохраняться при различного рода переходах состояния, и чтобы эта аксиома фрейма исполнялась сверху вниз, и, таким образом, ее применение не было бы восприимчивым к комбинаторному взрыву фактов, не имеющих отношения к цели, что характерно для рассуждений методом снизу вверх.

Изучение проблемы фреймов можно рассматривать как исследование доказуемости и именования. При идеальной реализации расширения списка (A, B, C) до списка (A, B, C, D) мы могли бы потребовать, чтобы система сама распознавала, что в процессе расширения сохраняются исходные элементы и соответствующие им номера позиций, а не вычисляла их каждый раз заново с помощью аксиомы фрейма. Фактически мы хотим, чтобы система распознавала и использовала (приблизительно) монотонный характер процесса расширения, в силу которого большая часть из того, что было доказано для старого состояния, остается справедливым и для последующего. Эта реализация могла бы также иметь возможность распознавать те (как правило, детерминированные) ситуации, когда следовало бы поддерживать только текущее состояние, передавая и его имя, и отведенную ему память его последователю. Такое поведение, разумеется, в точности совпадает с тем, которое традиционные программисты получают за счет использования деструктивного присваивания, хотя достигаемая при этом экономия компенсируется потерей семантической ясности. Ковалевский (1983б) указал пути, на которых этот механизм можно было бы незаметно извлекать из программы путем введения в нее соответствующих вызовов `demo`, оправдывающих реализацию деструктивного присваивания на основании того, что остается доказуемым при переходе в новое состояние. Пока, однако, не существует хорошо разработанной метауровневой формулировки, исходя из которой мы могли бы построить приемлемые стили программирования и достаточно эффективные методы реализации, приводящие к подобным механизмам обновления. Нахож-

дение такой формулировки считается исследовательской задачей первостепенной важности.

Наконец, логика как язык программирования, или, вернее, выступающий в этом качестве язык Пролог, часто подвергался критике за то, что он предоставляет пользователю слабые механизмы управления. В самом деле, для того, чтобы обеспечить желаемые действия в период исполнения, логические программисты вынуждены большей частью довольствоваться до некоторой степени произвольным и бессвязным набором управляющих арифманий и встроенных вызовов. В настоящее время общепризнаено, что усовершенствование применений метаязыка позволит в значительной степени свести на нет эту критику. Сообщения об управлении на метауровне исполнением логических программ можно найти в статьях Перейры (1982), а также Галлера и Лассера (1982).

Значение метаязыка хорионских дизъюнктов в более широком вычислительном контексте заключается в тех особых возможностях, которыми он обогащает программные средства благодаря амальгамированию. По своей важности роль этого метаязыка можно сравнить с ролью, играемой функциями более высокого порядка в управлении функциональными языками, такими как Лисп и НОРП. Свойство амальгамируемости присуще исключительно декларативным языкам программирования, что подкрепляет доводы, согласно которым декларативные языки будут в конечном счете доминировать над недекларативными формализмами.

VIII.2. Вычислительная практика

В этом разделе мы посмотрим, какое влияние логическое программирование оказывает на методологию программирования, и проиллюстрируем некоторые приложения, ставшие возможными благодаря его использованию

VIII.2.1. Методология программирования

Методология программирования охватывает те аспекты разработки программного обеспечения, которые связаны непосредственно с составлением программ для ЭВМ. В то время как многие «ремесленные» профессии развивали свои орудия для того, чтобы удовлетворять потребности своих практиков, положение в области программирования для ЭВМ было скорее обратным: языки программирования и инструментальные средства определялись главным образом заранее заданными особенностями архитектуры машин, а не рассмотрением наилучших способов

представления человеком вычислительных задач и методов их решения. Сами же эти особенности диктовались техническими и экономическими ограничениями в производстве логических схем.

Традиционные компьютеры, охватывающие первые четыре «поколения», сейчас обычно называют *машинами фон Неймана* в знак признания зависимости их от той модели механического вычисления, которая благодаря энергии и проницательности Джона фон Неймана воплотилась в физическую реальность первых компьютеров. В этой модели машинная операция представляет собой последовательность переходов состояний, действующих на дискретные ячейки памяти, причем каждый такой переход заменяет текущее состояние некоторой ячейки на новое. Цель каждого вычисления состоит тогда в том, чтобы преобразовать некоторое начальное состояние в некоторое конечное состояние, заставляя машину следовать заданной последовательности переходов. Программа нужна для того, чтобы точно определить эту последовательность, и она должна сама храниться в памяти машины. Программа, следовательно, должна состоять из дискретных инструкций, подробно описывающих как переходы состояний, так и их последовательность. Таким образом, мы приходим к понятию фон-неймановского языка программирования как языка инструкций для машины. Почти все используемые в настоящее время языки являются языками фон-неймановского типа; это относится к языкам Паскаль и Ада так же, как и к языкам Бейсик, Фортран и Кобол; это относится ко всем перечисленным языкам в той же мере, как и к языкам символьического ассемблера. Единственное существенное свойство, меняющееся вдоль спектра фон-неймановских языков, — это степень, в которой описание переходов состояний и управляющих ими решений абстрагируется от деталей конкретных типов реальных машин. Таким образом, нет никакой идейной разницы между командами ассемблера

```
LOAD I  
ADD ONE  
STORE I
```

для конкретной реальной машины и командой $I = I + 1$ языка Фортран для какой-либо абстрактной машины, которая с помощью подходящих интерпретирующих механизмов может быть реализована на реальной.

Машинно-ориентированный подход к программированию не вызывал бы возражений, если бы люди по природе своей могли свободно рассуждать о фон-неймановских процессах. Однако это, очевидно, не так: скольким программистам понравилась бы перспектива определить, можно ли произвольным образом вы-

бранное из самой середины фортрановской программы присваивание, не нанося ущерба, вычеркнуть или изменить? Причины, по которым эта задача оказывается столь устрашающей, часто перечислялись в кругах специалистов по информатике, но их стоит вновь повторить здесь для читателей, не знакомых с центральным аргументом, который состоит в следующем: роль каждого оператора в такой программе можно понять только в зависимости от контекста его исполнения, т. е. в зависимости от состояния машины в тот момент, когда устройство управления доходит до этого оператора. Само же состояние машины зависит от всей истории исполнения программы вплоть до данного момента.

Значение этого факта можно продемонстрировать очень просто. Представим себе, что где-то в одном месте программы содержится присваивание $S1: A := B + C$, а в другом месте — присваивание $S2: P := 10 * A$. Возможно, мы захотим провести анализ некоторого высказывания о значении P , вычисляемом посредством $S2$. Значение P , очевидно, определяется с помощью значения A , а последнее вычисляется посредством $S1$ как $B + C$. Можем ли мы на этом основании считать, что в нашем высказывании говорится о значении $10 * (B + C)$? Конечно, нет, ибо если управление доходит до оператора $S2$, то ранее оно могло и не проходить через $S1$, но даже если оно и прошло, то все равно значение A могло бы быть изменено другими операторами на пути от $S1$ до $S2$. Мы не можем этого знать без выделения и всестороннего анализа данного пути, что равносильно исполнению программы — и это всего-навсего исследовать один или два оператора, имеющих отношение к P !

Подстановка $(B + C)$ из $S1$ вместо A из $S2$ некорректна, поскольку она не сохраняет референта A — то, что A именует в $S1$, нельзя считать равным тому, что A именует в $S2$. Это — прямое следствие того, что указанные операторы являются командами деструктивного присваивания. Рассмотрим в противоположность этому логическую программу, содержащую два утверждения

$S1 : \text{родитель}(x, z) \text{ если } \text{мать}(x, z)$
 $S2 : \text{баб-и-дед}(x, y) \text{ если } \text{родитель}(x, z), \text{родитель}(z, y)$

И в $S1$, и в $S2$ предикат **родитель** (x, z) одинаково выражает принадлежность произвольной пары (x, z) отношению **родитель**, определяемому программой в целом. Мы можем, следовательно, подставить вместо предиката **родитель** (x, z) из $S2$ его референта **мать** (x, z), определяемого посредством утверждения $S1$, и получить

$S3 : \text{баб-и-дед}(x, y) \text{ если } \text{мать}(x, z), \text{родитель}(z, y)$

Данная подстановка будет корректной, поскольку утверждение S3 логически следует из S1 и S2. Логика и другие чисто декларативные языки обладают, стало быть, так называемой *референциальной прозрачностью*; то, на что ссылается каждая их компонента в программе, можно установить, используя корректность *подстановочности*, независимо от неуместного контекста исполнения программы. То, что предикат **баб-и-дед** в утверждении S3 ссылается на некоторую комбинацию предикатов **мать** и **родитель**, было определено путем рассмотрения того, на что ссылаются предикаты **баб-и-дед** и **родитель** в соответствии с S1 и S2.

Как это ни странно и ни печально, многие программисты еще не слишком хорошо осознали пагубное воздействие *деструктивного присваивания* на анализ программ, о котором говорилось в статье Бэкуса (1978). Отчасти причина этого состоит в том, что упомянутая операция глубоко укоренилась и считается сама собой разумеющейся с первых дней программирования для ЭВМ: действительно, некоторые программисты могут слабо осознавать, что есть и другие способы соотнесения данных с переменными. Другой причиной может быть то, что многие сторонники *«структурного программирования»*, начало которому положил Дейкстра (1976) в конце 60-х годов, чрезмерно сконцентрировали свое внимание на приведении в порядок конструкций управляющей логики программ, а не занимались серьезным изучением лежащих в основе семантик традиционных языков программирования, ошибочно считая, таким образом, что эти фундаментальные дефекты могут быть исправлены. Даже сегодня можно еще увидеть многочисленные публикации, посвященные бесполезной дискуссии о том, является ли один диалект языка Бейсик более «структурированным», чем другой. При имеющихся масштабах задач, стоящих в настоящее время в области управления программным обеспечением, эту деятельность можно, заимствуя выражение, характеризовать как спор о шезлонгах на палубе тонущего «Титаника».

Преимущества, достигаемые за счет альтернативного использования декларативных языков, давно, разумеется, были по достоинству оценены некоторыми группами программистов, в частности теми, что работают в области искусственного интеллекта. Быть может, одна из причин, в силу которых на эти языки обращали столь мало внимания в смутную эру структурного программирования, заключается в том, что в то же самое время, по крайней мере в Великобритании, разработки по искусственно му интеллекту испытывали принудительное сокращение. Как бы то ни было, эти языки стали с недавнего времени играть значительно большую роль в таких крупных проектах, как проект создания ЭВМ пятого поколения (Япония), программа ко-

митета Альви (Великобритания) и программа ESPRIT (европейское сообщество), отчасти благодаря осознанной теперь необходимости в повышении строгости и производительности процесса создания программного обеспечения, отчасти благодаря требованиям, поставленным в связи с разработкой новых архитектур вычислительных машин, и отчасти благодаря недавним успехам в области вычислительной логики и искусственного интеллекта. Тем не менее, даже до того как началась деятельность по разработке нового поколения компьютеров, развитие декларативных систем выдвигалось многими исследователями (см., например, сборник статей под редакцией Уоллиса (1982)) в качестве наиважнейшего условия для решения давно стоящих задач в области техники программного обеспечения.

Краткую оценку возможностей декларативных языков и соответствующих системных архитектур недавно дали Дарлингтон и Ковальский (1983). Они указали на существенные особенности, отличающие эти языки программирования от чисто процедурных языков, а именно на их двойственные функции — представления знаний и решения задач, спецификации и вычисления, а также на свойственный им параллелизм и благоприятное отношение к системам программного обеспечения, подвергающимся частым модификациям.

Нетрудно построить простые примеры, иллюстрирующие эти различия. Рассмотрим, например, следующее правило R , описывающее отношение друзья в базе данных D , состоящей из фактов вида x любит y :

$R : \text{друзья}(x, y) \text{ если } x \text{ любит } y, y \text{ любит } x$

Это правило осмысленно и само по себе как часть наших представлений о дружбе независимо от его возможных применений для решения задач. В недекларативных языках мы не можем даже сформулировать данное правило. Все, что мы сможем сделать, — это написать процедуру, поведение которой имитирует его конкретное применение, например, показать, что данная пара людей (x, y) — друзья. Если мы хотим обеспечить и другие применения, то нам потребуется написать еще ряд процедур, хотя они и не содержат никаких новых знаний по сравнению с тем, что определяется правилом R . Такая методология до смешного громоздкая и лишена гибкости.

Рассмотрим теперь спецификацию произвольного множества z друзей x в базе данных D :

$S : \text{быть-другом}(x, z) \Leftrightarrow (\forall y)(x \text{ любит } y, y \text{ любит } x \text{ если } y \in z)$

С помощью реализации полной стандартной формы языка логики предикатов первого порядка в качестве языка программи-

рования (см., например, Боузен, 1980) спецификация S может быть выполнена непосредственно, пусть и неэффективно, для решения вычислительных задач, связанных с базой данных D. Даже если такой подход окажется неудовлетворительным для больших объемов данных, сама его возможность дает нам средства быстрого моделирования на малых экспериментальных версиях баз данных. С другой стороны, спецификацию S можно использовать для вывода хорновских процедур

$$\begin{aligned} \text{быть-другом}(x, \emptyset) \\ \text{быть-другом}(x, y:z') \text{ если } x \text{ любит } y, \\ \quad y \text{ любит } x, \text{ быть-другом}(x, z') \end{aligned}$$

которые исполняются более эффективно. Перспективы исполнения логических спецификаций как программ недавно были изложены Ковальским (1983а).

Рассмотрим далее значение того факта, что для каждого конкретного x в базе данных определяются одновременно все индивидуумы из множества z друзей x . Очевидно, что при наличии достаточных ресурсов более естественно и более эффективно определять их параллельным образом, а не последовательным. Существующие параллельные архитектуры, такие как ALICE (см. Дарлингтон и Рив, 1983), могут извлекать подобное поведение из декларативных программ, не требуя, чтобы программист определял, как именно должно быть организовано их исполнение.

Дарлингтон и Ковальский (1983) утверждают, что эти отличительные особенности декларативных систем являются многообещающими для широкой области вычислительной деятельности. Например, коммерческие программы обработки данных можно было бы составлять прямо из правил, описывающих свойства данных; естественный и графический языки, которые по самой своей природе ведут к декларативному представлению, оказываются очень полезными для такой важной области, как человеко-машиинный интерфейс; декларативные представления форм, составных частей, компоновок, планов и целей проектирования могут, очевидно, многое дать для автоматизированного проектирования и производства.

Несмотря на то что сейчас прилагаются большие усилия, чтобы способствовать распространению осведомленности о таких возможностях и привлечь финансовые средства для их реализации, в настоящее время еще слишком рано говорить об обширном проспекте апробированных коммерческих или промышленных достижений, базирующихся на декларативных системах. Можно, однако, говорить об очень значительном про-

грессе в области интеллектуальных систем, основанных на знаниях; мы опишем эти важные приложения в следующем разделе.

VIII.2.2. Приложения

Первоначально логические программы использовались для представления и анализа подмножеств естественного языка. В значительной степени именно это приложение побудило Колмероэ и Русселя разработать первую реализацию Пролога. Вскоре после этого и другие исследователи в области искусственного интеллекта быстро нашли свои собственные приложения, первыми примерами которых были составление планов и написание компиляторов Уоррена (1977b), геометрические доказательства Уэлхэма (1976) и решение задач в механике Банди и др. (1979). С тех пор число сообщений о новых приложениях многократно возросло — например удивительная серия применений, найденных в Венгрии к 1980 г., была документирована Шантане-Тотом и Середи (1982). В дальнейшем мы сконцентрируем свое внимание на следующих важных классах приложений: *интеллектуальные системы, основанные на знаниях*, системы баз данных, экспертные системы и системы обработки естественного языка. Причина, по которой этим классам придается особое значение, состоит в том, что, как предсказывалось многими, именно они будут новыми принципиальными приложениями компьютеров в следующем десятилетии.

ИНТЕЛЛЕКТУАЛЬНЫЕ СИСТЕМЫ, ОСНОВАННЫЕ НА ЗНАНИЯХ (ИСОЗ)

Интеллектуальные системы, основанные на знаниях, — это такие системы, в которых механизмы интеллектуальных рассуждений применяются к явным представлениям знаний. Некоторые специалисты определяют область исследований, связанных с ИСОЗ, просто как прикладной искусственный интеллект. Системы баз данных, *системы, основанные на правилах*, и экспертные системы представляют собой различные, хотя и частично перекрывающие друг друга примеры ИСОЗ. Технология разработки ИСОЗ была выделена в отчете Альви (Великобритания, 1982) в качестве одной из четырех технологий, необходимых для эксплуатации в полном объеме следующего (пятого) поколения компьютерных систем. Тремя другими являются человеко-машинный интерфейс (ЧМИ), сверхбольшие интегральные схемы (СБИС) и технология программного обеспечения; как и в предыдущем случае, имеются значительные пересечения этих четырех областей. Взаимоотношения между ИСОЗ и новым поколением ЭВМ представляют собой фактически яркий

пример симбиоза: каждое из этих направлений необходимо для реализации полных возможностей другого. Тем не менее уже нынешнее состояние развития декларативных систем делает возможными многие полезные приложения ИСОЗ даже на малых традиционных компьютерах.

Следует подчеркнуть, что интеллектуальные системы, основанные на знаниях, нельзя получить просто в результате построения базы знаний и соответствующего механизма обработки информации для нее. Хотя фактически всякое составление, скажем, логической программы можно было бы справедливо назвать примером «программирования, основанного на знаниях», тем не менее неверно было бы говорить, что прологоподобная реализация этой программы обязательно образует ИСОЗ. Интеллектуальная обработка информации включает в себя, конечно, гораздо большие возможности, чем те, которые дают базисные средства построения вывода и выполнения поиска; по крайней мере для нее требуются стратегические механизмы (такие как эвристики), позволяющие сокращать ненужный поиск, используя, например, свою достаточную осведомленность о классе исследуемых задач. Интеллектуальные стратегии могут встраиваться непосредственно в интерпретатор, что дает весьма значительный эффект, когда эти стратегии являются достаточно общими. С другой стороны, прикладные программы объектного уровня могут вызываться посредством программ метауровня, которые содержат описания стратегий и которые при желании могут быть изъяты из интерпретатора и перепrogramмированы в соответствии с обстоятельствами. При таком подходе может быть использована в полной мере вся та сила и общность, которые дает амальгамирование объектного языка и метаязыка.

Слоумен (1983) указывает, что интеллектуальные системы, основанные на знаниях, играют более значительную роль, поскольку помимо того, что с их помощью в практику вводятся уже известные методы искусственного интеллекта, они и сами по себе могут помочь пролить свет на общие теории, связанные с представлением знаний и интеллектом, и способствовать их развитию. Он утверждает например, что ИСОЗ выполняют роль инструментов для исследования таких способностей, как умение совершать открытия, обучаться, способностей к творческой деятельности, общению, самоусовершенствованию и самоанализу, и предполагает, что может потребоваться развернуть эти системы в рамках целого ряда других структур представления знаний, а не только логики. Поскольку в настоящее время постулируется, что названными выше способностями должны обладать компьютеры нового поколения, мы можем сказать, что традиционные цели искусственного интеллекта становятся сейчас

актуальными для всего сообщества специалистов в области ЭВМ. Таким образом, мы можем, по-видимому, ожидать, что искусственный интеллект, и теперь остающийся далеко не малой областью информатики, в скором времени, быть может, объединит в себе гораздо большую ее часть.

СИСТЕМЫ БАЗ ДАННЫХ (СБД)

Наиболее отчетливым приложением ИСОЗ, которому в настоящее время уделяется много внимания, является их использование в *системах баз данных*. В обычных СБД данные традиционно рассматриваются как множества отношений, хранимых экстенсиональным образом в виде таблиц. Например, простая база данных, в которой регистрируются средние цены на лондонском рынке земельной собственности могла бы содержать следующие данные

<i>Челси, полдома, 7, полная земельная собственность,</i>	<i>£ 250 000;</i>
<i>Кенсингтон, жилая комната, 1, внаем, £ 65;</i>	
<i>Найтсбридж, квартира, 5, арендованная земельная</i>	
<i>собственность, £ 80 000;</i>	

и т. д.

В строках этой таблицы содержатся кортежи элементов вида
(область, тип, число комнат, вид собственности, стоимость)
которые все вместе составляют *n*-местное отношение.

Реляционная модель баз данных привела к реализации многих систем запросов посредством реляционных исчислений, в которых имеются стандартные реляционные операторы, такие как операторы соединения и проецирования. В традиционных СБД процессор обработки запросов обычно выводит из входного запроса некоторую специфическую конъюнкцию этих алгебраических операций, и затем для поиска отвечающих на запрос кортежей управляющая программа применяет полученные операции к таблицам.

Возможности использования логического программирования для представления данных и обработки запросов к базам данных были исследованы в ранних работах ван Эмдена (1978), Ковальского (1978) и Тернлунда (1978). Все они установили основной факт, согласно которому поиск данных в модели традиционной СБД содержится в стандартных механизмах построения выводов в логических интерпретаторах. В статье ван Эмдена этот факт прямо демонстрируется путем использования логики для переформулировки разработанной Злумом (1975) системы Query-by-Example. О формулировке этой системы на Прологе сообщается также в отчете Невиса и др. (1982).

Состояние искусства применения логики к базам данных вплоть до 1978 г. охватывается сборником статей под редакцией Галлера и Минкера (1978), тогда как более современные оценки вклада логики в эту область даются Далом (1981), Галлером (1981) и Ковальским (1981а).

В логической формулировке СБД поиск кортежей для ответа на запрос становится процессом доказательства теорем, в котором база данных (она могла бы быть просто множеством основных фактов) рассматривается как множество допущений, а запрос (некоторое целевое утверждение) — как теорема. Поиск кортежей в таком случае — это не что иное, как стандартный механизм извлечения ответа.

Помимо этих элементарных наблюдений следует отметить также и другие важные моменты, касающиеся использования логики в качестве языка баз данных. Главный из них заключается в том, что содержимое логической базы данных при отсутствии каких-либо допущений относительно внутренних представлений таблиц нет надобности ограничивать лишь основными фактами: база данных может включать также общие знания, представленные в виде правил, таких как

владение(*x*, РЕНТА) если тип(*x*, ЖИЛАЯ-КОМНАТА)

В традиционных (реляционных) СБД не имеется приемлемого способа включения подобных правил в саму базу данных — вместо этого их операционные действия должны быть закодированы в специально запрограммированных процедурах. Такого рода организация может стать тогда слишком неоднородной, поскольку в ней используются различные формальные системы для представления данных, правил, запросов и процедур анализа запросов. В логике же, как утверждается, все эти требования рассматриваются с единых позиций. В частности, в ней не делается различия между данными, представленными конкретными фактами, и данными, представленными в виде общих правил. Точно так же в ней не различаются правила, трактуемые как данные, и правила, трактуемые как процедуры. Таким образом, логика позволяет устраниТЬ часто упоминаемые и только мешающие искусственные различия между языками программирования, языками запросов и языками описания данных.

Фундаментальное несоответствие между реляционным подходом к СБД и подходом, основанным на доказательстве теорем, было охарактеризовано Никола и Галлером (1978). Они утверждают, что во втором подходе, где позволяет использовать обобщенные (посредством введения в них переменных) правила, база данных рассматривается как теория, обладающая многими возможными удовлетворяющими ей интерпретациями, в то время как в первом подходе, ограниченном работой лишь

с таблицами основных кортежей, она трактуется как одна конкретная интерпретация. Характерные недостатки реляционного подхода становятся тогда очевидными из тех трудностей, которые возникают как при организации рекурсивного доступа, так и при обработке неполностью определенных (частично означенных) кортежей в запросах и ответах.

Представляемые логикой преимущества иачниают приводить к отказу от подхода к реализации СБД, основанного на реляционных исчислениях, и в настоящее время ведутся многочисленные исследования, посвященные тому, как наилучшим образом заставить логику служить тем требованиям высокого уровня, которые предъявляются методологией СБД. Основное направление этих усилий связано с *оптимизацией запросов* и их анализом. Общепризнано, что языки запросов в СБД должны быть пригодны для употребления непрофессиональным пользователями, которые не обязаны самим иметь дело с механизмами хранения и поиска данных. С этой точки зрения запрос трактуется как спецификация, лишенная процедурных качеств. Задача разработчика состоит тогда в выборе наилучшего способа наложения запроса на базу данных. Одна из трудностей, возникающих при стандартном прологовском исполнении запросов, представленных целевыми утверждениями, заключается, как указывает Уоррен (1981), в том, что Пролог по умолчанию слепо следует текстуальному упорядочению подцелей в целевом утверждении, а это может оказаться чрезвычайно неэффективным способом. Еще хуже то, что в Прологе нет возможностей избавиться от избыточного недетерминизма: при задании запроса, скажем *найти(x)*, который сводится, допустим, к конъюнкции $p(a), r(a, y)$ с присваиванием $x := a$ (при отсутствии директив, вызывающих «отсечение») будет всякий раз выдаваться излишнее решение $x := a$ для каждого примера переменной y , удовлетворяющего отношению $r(a, y)$. Уоррен показывает, как эти недостатки могут быть до некоторой степени устранены в его СБД СНАТ-80 посредством введения в Пролог дополнительных возможностей, позволяющих планировать запросы (например, разумным образом переупорядочивая подцели) и оптимизировать запросы (замечая, например, независимость подцелей).

Дальнейшее исследование обработки запросов проводилось Чакраварти и др. (1982), которые продемонстрировали роль управления на метауровне в организации вычислений ответов на запросы. Они предложили интересный синтез подходов к реализации СБД, основанных на логическом выводе и реляционном исчислении, показав, как можно хранить накопленные в ходе дедуктивного вычисления ответов связывания в похожих на таблицы структурированных термах, внутренние представ-

лення которых могут быть сделаны чрезвычайно компактными при помощи схем, полностью аналогичных обычному методу совместного использования структур. Краткий отчет об этих и других родственных исследованиях, связанных с СБД и проводимых в Университете шт. Мэрленд, дается Минкером (1983).

Многими исследователями подчеркивалась также необходимость индексирования для организаций эффективной выборки данных из представленных в виде дизъюнктов. Возможно, что индексирование имеет даже большее значение для СБД, чем для обычных логических программ, которые, как правило, меньше заняты выборкой больших объемов основных данных. В контексте СБД, в частности, индексирование должно быть обычно ориентировано на данные, содержащиеся во вспомогательном, а не в первичном запоминающем устройстве. Методы эффективного индексирования, такие как методы *многоключевого хэширования* обсуждаются в статье Ллойда (1983).

Помимо этих чисто прагматических рассмотрений для исследования остается открытым целый ряд важных теоретических проблем, связанных с использованием логики в качестве языка баз данных. Из них числа мы можем упомянуть лишь знакомую проблему фреймов, возникающую при представлении времени, состояния и изменения, а также взаимодействие немонотонных рассуждений (например, рассуждений по умолчанию над неполностью определенным данным) с ограничениями целостности баз данных. Эти проблемы обсуждаются в статьях Ковалевского (1983c), Сергота (1983a) и Минкера (1981).

ЭКСПЕРТНЫЕ СИСТЕМЫ (ЭС)

Экспертная система представляет собой вид ИСОЗ, специально предназначенный для моделирования человеческих знаний и опыта в некоторой конкретной проблемной области. Как правило, ЭС будет обладать богатой базой знаний, составленной из фактов, правил и эвристик, относящихся к этой области, и возможностями в диалоговом режиме проводить консультации со своим пользователями почти так же, как это мог бы делать эксперт-человек. Кроме того, она должна быть в состоянии объяснять пользователю все предлагаемые ей советы или решения, а также, быть может, иметь способности к совершенствованию своего искусства путем использования опыта, накопленного в процессе такого взаимодействия. Подробное описание свойств и предполагаемых сфер применения ЭС дается в книге, вышедшей под редакцией Минч (1979).

Экспертные системы образуют очевидную и очень привлекательную область приложений для логического программирования. Роль, которую может играть логика в этой области, хорошо освещается в статье Ковалевского (1983d). Он подчеркивает тот

факт, что ЭС следует отличать от простых систем, основанных на использовании правил, которым часто неверно приписывают свойства экспертных исключительно на основании того, что они содержат факты объектного уровня и правила, адекватно представляющие некоторое подмножество экспертных знаний. Различия заключаются в том, что эксперт-человек оказывается в состоянии призвать себе на помощь — вследствие накопленного опыта, вдохновения или интуиции — разнообразные специфические для данной области эвристики, позволяющие ему прокладывать разумные пути в огромных пространствах поиска и реагировать согласно здравому смыслу на всякого рода несовместимости и неполноту, которые он осознает в процессе развертывания своих знаний. Короче говоря, зная факты и правила, относящиеся к проблемной области, он, кроме того, понимает, какими эффективно воспользоваться.

Последнее обстоятельство лежит в основе тезиса Ковальского, согласно которому человеческие экспертные знания обычно не поддаются полной формализации (являющейся, в некотором отношении, их заменой), и поэтому нет оснований надеяться, что эти знания будут иметь точную спецификацию, которой они должны соответствовать. Следует, таким образом, ожидать, что процесс введения экспертных знаний в программу для компьютера — в противоположность точной разработке хорошо специфицированных обычных программ — будет протекать посредством проб и ошибок. По мере того как база знаний развивается и к компетентности системы предъявляются новые требования, в нее должны вноситься соответствующие корректиды — точно так же и спецификацию программ возможно потребовалось бы совершенствовать для того, чтобы внести поправки и уточнения и привести ее в соответствие с новыми потребностями. По своей природе логическое программирование более, чем какой-либо другой из существующих вычислительных формализмов, приспособлено для решения задач обнаружения противоречивости (например, с помощью процедур опровержения) и неполноты (например, посредством правил вывода по умолчанию) с целью мотивации необходимости усовершенствования экспертной системы, в то время как метаязык логического программирования обеспечивает автоматическое описание критериев и механизмов для этого усовершенствования.

Логика как язык экспертных систем исследовалась Хаммондом (1980). Дальнейшее развитие этот вопрос получил в статье Кларка и Маккейба (1982), которые определили ряд основных требований к ЭС: они должны обладать системой логического вывода в качестве базиса для обработки запросов, способностью пополнять базу знаний той информацией, которая уже была получена (порождение лемм), способностью объяснять пользова-

телю, как и почему было выведено каждое конкретное заключение, и способностью получать знания от человека (принцип «диалоговой симметрии»). Кларк и Маккейб реализовали данные требования главным образом путем добавления специальных управляющих возможностей к правилам объектного уровня, содержащимся в базе знаний. Впоследствии более сложный способ реализации указанных требований посредством использования метаязыка был описан Хаммондом и Серготом (1983). В их системе устройство Query-the-User комбинируется с оболочкой экспертных систем (APES). Эти составные части описаны в предыдущих отчетах Сергота (1983b) и Хаммонда (1983a) соответственно. Система реализована на микро-Прологе и дает очень гибкий инструмент, называемый APE-the-User и предназначенный для построения экспертных систем. Пользователь-эксперт может ввести какое-либо утверждение, объявляющее, что он готов ответить на вопросы, касающиеся некоторого отношения. После этого система рассматривает эксперта как расширение имеющейся в ней внутренней базы знаний и задает ему вопросы с тем, чтобы извлечь дополнительную информацию об указанном отношении, которая затем будет храниться в базе знаний. Таким образом осуществляется кооперация системы и эксперта с целью построения базы знаний. Запросы пользователя на объектном уровне могут обрабатываться непосредственно путем обращения к интерпретатору микро-Пролога, расположенному в ядре системы, в то время как разнообразные запросы типа «как», «почему» и «почему не», требующие поиска объяснения, обрабатываются посредством обращения к интерпретатору метауровня (который сам исполняется с помощью микро-Пролога); он в свою очередь строит и выдает объяснения в виде отредактированных доказательств. В системе предусмотрены также средства для структурирования запросов на естественном языке. Сочетание всех этих возможностей образует благоприятную универсальную среду для построения, модификации и эксплуатации экспертных систем.

Система APES была с успехом использована для автоматизации юридических экспертных систем. Хаммонд (1983b) описывает, например, как в сотрудничестве с министерством здравоохранения и социального обеспечения в базе знаний системы APES были собраны более двухсот недискреционных инструкций, регулирующих норму дополнительного пособия. В более широких масштабах и при наличии гораздо больших юридических и лингвистических сложностей система APES была использована для кодирования принятого в 1981 г. закона о гражданстве в Великобритании, который определяет категории британского гражданства и включает правила, предназначенные для своей же собственной интерпретации. В своем отчете об этой

работе Кори и др. (1984) отмечают два интересных момента относительно логической формализации права: во-первых, так как все законы записываются таким образом, чтобы при этом обеспечивалась точность и полнота (в юридическом смысле), то оказывается возможным избежать многих известных трудностей, связанных с получением знаний от эксперта-человека; во-вторых, поскольку законодательство может рассматриваться как специфическая категория сложного программного обеспечения, опыт его компьютеризации может способствовать более глубокому пониманию общих проблем техники программного обеспечения. Еще один пример использования Пролога для представления законов был описан Хастлером (1982) в связи с законом об оскорблении действием, а более общий взгляд на роль логики в юриспруденции дается Серготом (1982).

Собрание имеющихся на сегодняшний день документальных свидетельств использования логического программирования для технологии экспертных систем содержится в материалах конференции, организованной Британским обществом информатики (1983); в статьях некоторых исследователей объясняются обнаруженные ими преимущества применения логики перед исторически более популярным выбором Лиспа и приводятся сравнения с такими известными основанными на Лиспе системами, как экспертная система медицинской диагностики EMYCIN. Экспертные системы, основанные на логике, разрабатываются также в Лиссабонском университете: о системе ORBI, предназначенней для оценки ресурсов окружающей среды, сообщается Перейрой и др. (1982).

ОБРАБОТКА СООБЩЕНИЙ НА ЕСТЕСТВЕННОМ ЯЗЫКЕ (ЕЯ)

Обработка сообщений на ЕЯ играет очень важную роль в разработке средств для обеспечения человека-машинного интерфейса и, в частности, в построении внешних уровней для ИСОЗ. Она представляет собой, следовательно, важную область приложения логического программирования.

Для реализации естественного языка на ЭВМ требуется формализовать как его синтаксис, так и семантику. Использование с этой целью логики хорновских дизъюнктов первым начал изучать Колмероэ, который впоследствии работал в сотрудничестве с Ковалевским (1974а). Они показали, что хорновских дизъюнктов оказывается достаточно для выражения произвольной *контекстно-свободной грамматики* (КСГ), что вопросы относительно структуры предложений ЕЯ можно формулировать как целевые утверждения и что различные процедуры доказательства, применяемые к логическим представлениям ЕЯ, соответствуют различным стратегиям синтаксического анализа,

Грамматику хорновских дизъюнктов обычно называют *грамматикой определенных дизъюнктов* (ГОД), поскольку хорновские дизъюнкты — факты и импликации — известны как определенные дизъюнкты. Дескриптивные и операционные свойства этих грамматик и их соотношения с другими формализациями ЕЯ исследовалась Перейрой и Уорреном (1980).

Сущность естественного языка такова, что многие из его свойств лучше всего представлять с помощью *контекстно-зависимых грамматик*, т. е. грамматик, содержащих правила, согласно которым определенные структуры выражений классифицируются в зависимости от контекста их вхождения в рассматриваемые предложения ЕЯ, а не в зависимости лишь от их собственного строения, как это делается в КСГ. Впервые логика хорновских дизъюнктов для представления контекстной зависимости была использована Колмероэ (1978) в процессе разработки «*метаморфозных грамматик*», для которых грамматики определенных дизъюнктов образуют нормальную форму. Колмероэ (1982) приводит логическую формулировку в виде этих грамматик одного полезного подмножества ЕЯ, показывая, в частности, как в строго классической логике могут представляться расширенные понятия квантификации с тем, чтобы отличать семантически осмыслившиеся предложения естественного языка от бессмысличных. Обзоры работ по грамматикам определенных дизъюнктов и другим родственным грамматикам недавно написаны Перейрой (1983) и Абрамсоном (1983).

Более общей структурой представления, приспособленной как для естественноязыковых, так и для других конструкций, являются семантические сети. Делияни и Ковальский (1979) показали, что традиционную формулировку семантических сетей можно было бы обобщить с тем, чтобы представлять множества предложений в виде дизъюнктов. В этом обобщении узлы сети представляют термы, а соединяющие узлы дуги представляют консеквентные или антecedентные бинарные отношения между ними; множество всех дуг, выходящих из каждого узла, представляет тогда некоторый дизъюнкт. Эти системы обеспечивают очень компактное и единообразное представление, и, кроме того, к ним можно применять процедурные интерпретации, дающие операционные схемы извлечения информации из таких сетей. Обсуждение основанных на логике семантических сетей и подкрепляемого ими тезиса «ЕЯ = логика + управление» можно найти в книге Ковальского (1979а).

VIII.2.3. Обучение

Логическое программирование обещает сделать значительный вклад в применение компьютеров с целью обучения. Впервые этот проект был проверен в 1978 г., когда Ковальский ввел

предмет логического программирования в средней школе Park House в Уимблдоне, используя прямой доступ к вычислительным средствам, находящимся в Имперском колледже. Успех проводившихся занятий был таков, что в 1980 году при поддержке научно-исследовательского совета был принят более обширный проект «Логика как язык ЭВМ для детей». Первоначальные цели, материалы и методология программы обучения описаны Энналсом (1980), тогда как более современная оценка этой работы дается Энналсом (1982).

В настоящее время данный проект основывается на реализации микро-Пролога. Дети в возрасте 10—12 лет обучались микро-Прологу по несложным этапам, продвигаясь от простых баз данных, содержащих только факты без переменных, запросы к которым состояли лишь из одного вызова, к употреблению процедур и запросов общего вида. Изображения семантических сетей способствовали объяснению рекурсии, в то время как удобное представление списков, обеспечивающее микро-Прологом, позволило ввести понятие структуры данных.

Логическое программирование может применяться не только для знакомства детей с вычислениями на ЭВМ. Его можно использовать также для обогащения преподавания всех академических предметов из школьного учебного плана. Эти возможности иллюстрируются в различных статьях Энналса (1981). Он указывает, что логика является единственной академической дисциплиной, общей со всеми школьными предметами, поскольку она везде способствует ясности понимания и изложения. Среди многих приводимых им примеров применения логики для преподавания других предметов имеются игры, моделирующие исторические события, на уроках истории, молекулярный анализ на уроках естествознания, решение уравнений на уроках математики и грамматические правила на уроках французского языка. Употребление для этих целей микро-Пролога позволило также использовать такие построенные с его помощью средства, как система *Query-the-User*, разработанная Серготом (1983b); использование этой системы для обучения детей правильной формулировке логических запросов исследовано Уэйром (1982). Бриггс (1984) описал недавно конструкцию нового интерфейса с микро-Прологом, в котором внешний синтаксис специально адаптирован с целью легкого усвоения его учениками младших классов; фактически логическому программированию обучаются сейчас дети в возрасте 7-9 лет.

Бурный рост персональных или любительских вычислений в наши дни таков, что многие программисты-любители являются, к счастью или несчастью, самоучками, не получившими формального образования в области программирования. В настоящее время в соответствии с текущим состоянием рынка про-

граммного обеспечения персональных ЭВМ они ориентируются главным образом на язык Бейсик. Однако наличие готового микро-Пролога и превосходного самоучителя программирования на нем, написанного Кларком и Маккейбом (1984), может привести к изменению этой ситуации.

Преподаватели логического программирования соглашаются, как правило, в том, что логику более легко воспринимают те, у кого нет предшествующего опыта вычислений на машине, чем те, которые уже привыкли к традиционным формальным системам программирования. Того, кто в течение десяти лет програмировал с помощью такого формализма при поддержке хорошо знакомого обеспечения и с ожиданием определенных стандартов, потребуется, видимо, долго убеждать в том, что логика, находящаяся все еще на сравнительно ранней стадии развития, предлагает довольно заманчивую альтернативу. Разумеется, нельзя ожидать — или даже желать — чтобы обращение в другую веру состоялось сразу же вслед за демонстрацией хорошо подобранных примеров, которые тонко показывают в выгодном свете такие фантастические возможности, как получение многочисленных решений или свойство обратимости. Более общие принципы методологии программирования, важность приложений, связанных с базами знаний, и соответствующая эксплуатация грядущего нового поколения архитектур вычислительных машин — вот те действительные проблемы, которые следует использовать для мотивировки интереса к логике, и все эти проблемы, где необходимо, нужно довести до сознания при помощи надлежащей подготовки и образовательных программ. До сих пор, однако, имеется слишком мало доступной литературы, написанной специально для того, чтобы помочь работающим профессиональным программистам по достоинству оценить логику. Некоторые из проблем, к которым следовало бы обратиться такого рода литературе, особенно в связи с концептуальным переходом от процедурного детерминизма к декларативному недетерминизму, обрисованы в статьях Бирда (1980), а также Клужняка и Шпаковича (1982).

VIII.3. Вычислительная техника

Следующее поколение компьютеров проектируется для того, чтобы охватить широкий диапазон мощных возможностей обработки информации, которые в принципе не осуществимы на наших современных фон-неймановских машинах. К ним относятся, в частности, схемы редукционного исполнения и исполнения под управлением потока данных, которые выявляют и используют параллелизм, свойственный многим программам для ЭВМ.

В новых вычислительных машинах такие схемы будут реализованы на основе мультипроцессорной архитектуры с высокой степенью распараллеливания, что приведет к разительному увеличению мощности обработки информации по сравнению со стандартом нынешних вычислений. Такого рода машины на самом деле, конечно, уже существуют; по-видимому, наиболее известной в Великобритании является манчестерская машина, управляемая потоком данных, характеристики которой описаны Гардом и Уотсоном (1980).

Тем не менее, чтобы обеспечить преимущества, достигаемые за счет применения схем параллельного исполнения, недостаточно лишь создать новые вычислительные машины. Мы можем, в конце концов, уже сейчас реализовать их, просто соединив параллельно несколько фон-иеймановских компьютеров. Причина, по которой такой подход оказывается не в состоянии дать желаемого уровня производительности, заключается в характере каждой отдельной фон-неймановской машины. Этот вид машин предполагает выполнение детерминированной, директивно управляемой последовательности присваиваний в первую очередь над скалярными числовыми данными; он предлагает в принципе неблагоприятную среду для реализации недетерминированных, управляемых логическим выводом или данными одновременных вычислений, и в особенности эта среда непригодна для обработки структурированных данных нечисловой природы. Чтобы избавиться от этих ограничений, требуется отказаться не только от машин фон Неймана, но также и от базирующихся на них языков программирования. Именно поэтому в настоящее время столь много внимания фокусируется на декларативных языках, возможности которых совместимы с широким диапазоном новых схем вычислений и архитектур вычислительных машин. Эти языки непосредственно способствуют прогрессу нового поколения вычислительной техники благодаря тому, что делают его использование осуществимым.

VIII.3.1. Логика как нефон-неймановский язык программирования

Основным свойством, в силу которого логика и другие декларативные языки становятся независимыми от фон-иеймановского понятия вычисления, является их семантическая нейтральность по отношению к стратегии исполнения. Ранее мы указывали уже на это свойство как на полное отделение логики от управления: если какая-то задача решается на компьютере с помощью логической программы, то это оказывается возможным исключительно благодаря тем следствиям, которые позволяет вывести логическое содержание программы, а не из-за способа

исследования их компьютером. Возможности, которые содержатся в этом наблюдении для сопрограммных и параллельных схем исполнения логических программ давно уже были обнаружены Ковальским (1979а).

Стандартная стратегия Пролога была задумана таким образом, чтобы с учетом относительной простоты ее реализации она давала программисту некоторые возможности управления выбором вызовов и выбором процедур в условиях эксплуатации единственного процессора. Первые практические усилия по реализации более свободной стратегии концентрировались на исполнении в *сопрограммном режиме, управляемом потоком данных*, которое можно рассматривать как обобщение стандартного правила вычисления (правила выбора вызова). Согласно этой схеме активация вызовов управляет потоком данных через содержащиеся в вызовах общие переменные, а не предписанной заранее или неявно заданной управляющей последовательностью. Исполнение в сопрограммном режиме является главной особенностью управления в системе ИС-Пролог, описанной Кларком, Маккейбом и Грегори (1982) и разработанной на основе предшествующих исследований Стивенса (1977) *схем ленивого вычисления* в Прологе. В ИС-Прологе какая-либо переменная из входной программы, скажем переменная x , может быть аннотирована одним из символов «?» или « \wedge », указывающим на зависимость способа исполнения содержащего ее вызова от состояния связности x в точке активации. Более точно, наличие в вызове аннотации $x?$ определяет, что управление должно сразу же переходить к этому вызову всякий раз, когда на некотором шаге исполнения программы либо переменной x присваивается значение, либо значение присваивается какой-то другой переменной, входящей в терм, уже связанный с x — иными словами, всякий раз, когда исполнение программы увеличивает степень конкретизации вычисленного значения переменной x (или, проще говоря, передает данные переменной x). В этой точке текущее (возможно, незавершенное) исполнение какого бы то ни было обрабатывавшегося ранее вызова приостанавливается, а интерпретатор тем временем приступает к обработке того вызова, к которому только что было привлечено его внимание. Интерпретатор продолжает вычисления по этому пути до тех пор, пока он не увидит, что на следующем шаге исполнения переменной x требуется передать какие-то данные, после чего выполнявшееся вычисление приостанавливается, а управление вновь возвращается к предыдущей отложенной точке. Другая возможная аннотация $x\wedge$ означает, что управление должно сразу же переходить к содержащему ее вызову, приостанавливая текущее вычисление, всякий раз, когда интерпретатор предвидит, что на следующем шаге исполнения дан-

ные требуется передать переменной x ; впоследствии управление вернется к этой отложенной точке, лишь когда на некотором шаге исполнения в новом вычислении данные будут успешно переданы x . Таким образом, действия двух этих аннотаций, грубо говоря, дополняют друг друга. При указанной схеме управление просматривает пространство вычислений более произвольным образом, чем это делается при стандартной стратегии Пролога: говоря неформально, оно перепрыгивает с одного частично завершенного вычисления на другое в соответствии с потоком данных через аннотированные переменные. В упомянутой ранее статье Кларка и др. даются иллюстрации повышения эффективности исполнения программ, которое достигается за счет использования этого более богатого вида поведения.

Поскольку вызов, содержащий $x?$, привлекает к себе внимание, как только на некотором шаге исполнения переменной x будут переданы дополнительные данные, его называют *страстным потребителем* (данных для x), а поскольку вызов, содержащий x^A привлекает к себе внимание только тогда, когда на некотором шаге исполнения потребуются его возможности предоставить дополнительные данные для x , его называют *ленивым производителем* (данных для x). *Протокол производителей-потребителей* лежит в основе большинства моделей исполнения в сопрограммном режиме под управлением потока данных. Относя различными способами выбранные вызовы программы к той или другой категории, можно получить широкий спектр методов исполнения, простирающийся от совершенно ленивого производства данных до абсолютно страстного их потребления.

Приблизительно в то же самое время, когда разрабатывался 1С-Пролог, Перейра и Монтеиро (1981) предложили другой интересный подход к построению логических интерпретаторов для сопрограммного и параллельного исполнения программ путем определения этих видов поведения в самой логике хорновских дизьюнктов. Они, таким образом, пришли к описаниям управления на метауровне, которые при исполнении функционируют как промежуточные интерпретаторы, сами способные извлекать эти виды поведения из стандартных логических программ объектного уровня. Однако вместо того, чтобы использовать для достижения координации аннотации потока данных, они воспользовались явными системными вызовами, обладающими разнообразными способностями запрашивать состояния связности переменных и заставлять процесс приостанавливаться.

Исполнение в сопрограммном режиме поглощается более общим понятием *исполнения под управлением потока данных*, которое осуществляется на основе работы с хранимым в памяти графовым представлением зависимостей по

данным между операциями во входной программе. Каждая вершина этого графа представляет собой некоторую операцию. При всякой ее активации поглощается одна или несколько меток данных, поступающих по входящим в вершину дугам, с тем чтобы получить (путем выполнения соответствующей вершине операции) новую метку данных, которая передается по всем выходящим из вершины дугам. Соединяющие вершины дуги служат, таким образом, в качестве односторонних каналов, по которым осуществляется передача данных в графе. Сам граф должен быть скомпилирован из входной программы посредством определения взаимозависимостей между ее операциями; очевидно, что проще всего эту задачу решать для языков, обладающих референциальной прозрачностью.

При исполнении программы граф ведет себя подобно сети процессоров, согласованно производящих и потребляющих данные. Для реализации простых вычислительных механизмов граф может оставаться топологически статическим в течение всего исполнения, однако для обеспечения других механизмов — в особенности рекурсии — могут потребоваться либо динамическая модификация графа, либо более сложные схемы маркирования.

Такая схема исполнения дает значительные возможности для распараллеливания, поскольку каждая вершина графа может быть активирована для выполнения соответствующей ей операции в любой момент после того, как она будет разблокирована в результате поступления всех необходимых входных данных. При самом грубом подходе можно просто произвести одну или несколько начальных активаций, а затем предоставить полную свободу всем имеющимся в сети процессорам, вовсе не пытаясь скординировать получающиеся в результате процессы. Неудивительно, что в этом случае исполнение программы становится чрезвычайно беззащитным по отношению к различным нежелательным неустойчивым состояниям — одной из возможностей, например, является насыщение каналов, которое происходит, когда в одном или нескольких каналах данные поставляются быстрее, чем потребляются, что вынуждает временно приостанавливать работу разблокированных производителей и тем самым не полностью использовать имеющиеся возможности по обработке данных до тех пор, пока это насыщение не будет ликвидировано.

Одну возможную потоковую модель исполнения логических программ предложили ван Эмден и де Лусена Фило (1982). Сначала они показали, как с помощью хорновской процедуры можно специфицировать отношение, вычисляемое в той или иной вершине графа потока данных; связь между двумя любыми вершинами графа представима тогда в виде логического

запроса, соединяющего два вызова процедуры, и активацию процесса можно интерпретировать как активацию вызова в этом запросе. В результате им удалось в противоположность обычной (последовательной) процедурной интерпретации логики сформулировать (параллельную) *интерпретацию посредством процессов*. Та же самая потоковая модель в последующей статье Брафа и ван Эмдена (1984) была связана с логическим представлением традиционных блок-схем, разработанным ранее Кларком и ван Эмденом (1981).

Исполнение под управлением потока данных — это лишь одна из многих схем, предназначенных для использования параллелизма между индивидуальными процессами. Более того, всякая данная формальная система программирования может допускать различные виды параллелизма, каждый из которых предполагает наличие особого способа управления реализующими эту систему процессами. Возможности для параллельного выполнения логических программ распадаются на несколько категорий. К первой категории, оказавшейся довольно приемлемой с точки зрения разработчика реализации, относится *ИЛИ-параллелизм*, в котором используется тот факт, что когда на активируемый вызов отвечают сразу несколько процедур, их можно вызывать и применять для исследования этого вызова параллельно. Получающиеся в результате параллельные вычисления можно строить независимо, за исключением того случая, когда у них появляются ссылки на одни и те же несвязанные переменные в их общей родительской среде, вследствие чего могут возникнуть конкурирующие присваивания; для устранения этого незначительного препятствия на пути к полному независимому ИЛИ-параллелизму можно использовать специальные схемы организаций стеков. ИЛИ-параллелизм должен сыграть полезную роль при поиске в базах данных, где один запрос может иметь большое число альтернативных решений; это — прямое следствие недетерминизма логических программ в том смысле, что они допускают целое множество вычислений.

Ко второй категории относится *И-параллелизм*, в котором используется тот факт, что содержащиеся в запросе вызовы можно активировать и решать параллельно. И-параллелизм является главной особенностью схем решения задач посредством согласованно действующих «параллельных процессов». Эффективность этих схем зависит от точности управления передачей данных между процессами (например, посредством имеющихся в вызовах общих переменных) и синхронизацией процессов (например, посредством установления приоритетов связываний). Действительно, независимый И-параллелизм возможен только в том случае, когда параллельные вызовы не содержат общих переменных. Однако общую переменную, такую

как переменная x в целевом утверждении $?p(x), q(x)$ можно заставить служить различным полезным целям при посредничестве между не полностью независимыми параллельными исполнениями вызовов, содержащих эту переменную. В тех ситуациях, когда программист не желает сам заботиться о координации вызовов и не предлагает интерпретатору никаких руководящих инструкций по этому поводу, общая переменная в вызовах может представлять некоторую проблему для разработчика реализации. С логической точки зрения вызовы p и q должны быть «согласованы» относительно присваивания переменной x какого-либо значения, и проблема состоит в том, чтобы достичь этого согласования операционно, сохранив насколько возможно параллелизм исполнения программы. Значение данного требования сильно возрастает, когда либо p , либо q , либо оба этих вызова могут быть решены недетерминированно, поскольку согласование p и q следует искать тогда в виде пересечения их индивидуальных множеств решений для переменной x . Для уменьшения указанных трудностей обычно используется компромиссный вариант: один из И-параллельных вызовов назначается производителем общей переменной, а все остальные — потребителями. Это делается, исходя из того, что, по всей вероятности, нет достаточных оснований, относящихся к области решения задач, иметь более одного производителя. При другом подходе многочисленные решения И-параллельных вызовов могут вообще не согласовываться (и тем самым потенциально приносится в жертву полнота) с целью получения более простой организации распределения ресурсов обработки данных.

К третьей категории относится *параллелизм с использованием потоков* (или «конвейерная обработка структур»), в котором вызову разрешается начать потребление каких-либо структурированных данных например, списка, производимых другим вызовом, как только первый вызов получит некоторую подструктуру этих данных. Тем временем вызов-производитель будет параллельно продолжать порождение следующей подструктуры. Эту категорию можно рассматривать как особый способ обращения с И-параллелизмом при наличии общих переменных. Она дает превосходное средство для достижения координации процессов, основанное на протоколах передачи сообщений, особенно в тех случаях, когда передаваемым подструктурам разрешается содержать неконкретизированные переменные, пригодные для двусторонней связи.

На более мелких шагах параллелизма можно достигнуть путем реализации его, насколько это возможно, в алгоритме унификации. Это может привести к значительному повышению эффективности даже последовательного Пролога. Точная схема такого параллелизма была описана Тиком и Уорреном (1984), по

оценкам которых потенциальная производительность при этом приближается к 450 клвс (1 клвс = 10^3 логических выводов в секунду)¹⁾, что дает по крайней мере 20-кратное увеличение по сравнению с эффективностью реализации Пролога для машины DEC-10. Они предлагают разложить компилируемые из входных логических программ прологовские инструкции высокого уровня на совмещенные по времени выполнения микрокоманды, обрабатываемые посредством *конвейерного параллелизма*.

Этн и другие разновидности параллелизма были классифицированы и исследованы Конерн и Кнблером (1981), которые разработали «модель И/ИЛИ процессов» для параллельного исполнения логических программ. В указанной модели обеспечивается главным образом ИЛИ-параллелизм, но предлагается также и ограниченная форма И-параллелизма — независимые вызовы исполняются действительно И-параллельно, а для того, чтобы обрабатывать вызовы с общими переменными, они используют упомянутый ранее метод, посредством которого один из вызовов делается производителем, а все остальные — потребителями; вызов-производитель и вызовы-потребители исполняются в этом случае последовательно в соответствии с порядком, определяемым интерпретатором на основании анализа потока данных.

Детальные исследования схем для ИЛИ-параллелизма ранее были выполнены в докторских диссертациях Полларда (1981) и Конерн (1983). Как мы уже отмечали, основные трудности реализации этих схем сосредоточены в организациии связываний в ИЛИ-параллельных подвычислениях. Существует, например, проблема организациии действий, связанных с минимизацией избыточности среди того, что каждое из этих подвычислений наследует из текущей среды при спуске из активирующей родительской вершины. Имеется также тактическая проблема организациии управления связыванием и восстановлением («прослеживанием») переменных в той среде, к которой ИЛИ-параллельные процессы имеют совместный доступ. Один интересный подход к решению последней проблемы объясняется в статье Боргварда (1984). В диссертации Полларда рассматриваются также возможные методы идеального полного обеспечения И/ИЛИ-параллелизма (в котором комбинируются оба эти вида параллелизма). Он исследует развертывание отдельных одновременных процессов, ответственных за управление порождением и согласованием связываний. Никаких практических схем такого подхода до сих пор реализовано не было.

¹⁾ Под логическим выводом здесь понимается одно применение правила резолюции (получение одной резольвенты). — Прим. перев.

В последние годы большой интерес вызвали две реализации: *Парлог*, разработанный Кларком и Грегори (1984), и *Concurrent Prolog (CP)*, разработанный Шапиро (1983б). *Парлог* представляет собой расширение параллельного реляционного языка, разработанного ранее Кларком и Грегори (1981). В нем обеспечиваются оба вида И- и ИЛИ-параллелизма, тогда как в СР обеспечивается в настоящее время лишь один И-параллелизм. В обеих этих системах обеспечивается, кроме того, *неопределенность выбора процедуры*, а именно в каждой процедуре пользователю разрешается определять какое-либо подмножество вызовов из ее тела, которое будет служить в качестве *охраны* данной процедуры. Согласно этой схеме, если на некоторый вызов отвечают несколько процедур, то их охранные части, когда они имеются, исполняются параллельно; та процедура, чья охранная часть решается первой, подается затем активирующему вызову, а остальные процедуры пока не используются. В этой схеме внутренне заложена неопределенность, поскольку конкуренция между потенциально успешными охранными управляет распределением машинных ресурсов, а не логикой программы или указаниями пользователя. В обеих системах предусмотрен также широкий диапазон механизмов управления параллельными процессами. Поведение *Парлога* управляется производным в период компиляции анализом декларативных видов входа и выхода (подобных тем, что допускаются в реализации Пролога для машины DEC-10, однако несколько более сильных), а также разнообразными уточняющими аннотациями, которые пользователь может помещать на выбранных им переменных. В СР же, с другой стороны, используется лишь единственный механизм помещения на переменных аннотаций «только считывание», который позволяет достигать почти тех же самых целей, но при помощи существенно отличающегося стиля программирования. Перспективы реализации *Парлога* на системе ALICE, которую отличает *параллельная архитектура редукционного графа*, описаны Дарлингтоном и Ривом (1981, 1983).

Критическое сравнение СР с реляционными языками, предшествовавшими *Парлогу*, дается Шапиро (1983б). В свою очередь сравнение *Парлога* и СР проводится Кларком и Грегори (1983). Хеллерстайн и Шапиро (1984) ярко продемонстрировали элегантность и силу СР, применяв его к чрезвычайно сложным параллельным алгоритмам решения задачи максимизации потоков в сетях и показав, что имеющиеся в СР средства передачи сообщений высокого уровня позволяют извлекать производительность такого же порядка, который достижим в традиционных языках с помощью механизмов присваивания машинного уровня. Дополнительную информацию относительно *Парлога*,

СР, а также другие предложения, касающиеся параллельного логического программирования, можно найти в материалах Симпозиума в Атлантик Сити (1984).

VIII.3.2. Проект создания ЭВМ пятого поколения

В 1979 г. министерство внешней торговли и правительство Японии санкционировали исследования возможности осуществления нового и исключительно претенциозного предприятия в области вычислительной техники, которое обычно называют проектом создания *систем ЭВМ пятого поколения*. Национальная цель этого проекта заключается в том, чтобы вывести Японию на ведущие, новаторские позиции в области новой компьютерной технологии, а главная техническая цель, выполнение которой распланировано на десятилетие вперед, состоит в соединении высокопараллельной архитектуры вычислительных машин, отличной от архитектуры фон Неймана, со схемами обработки знаний, разработанными в результате исследований в области искусственного интеллекта.

Цели проекта были обнародованы на международной конференции, состоявшейся в японском научно-исследовательском центре по обработке информации (JIPDEC, 1981), и работы по нему начались с весны 1981 г. под руководством Кацухиро Фучи. Они ведутся в основном в специально организованном для этих целей Институте нового поколения вычислительной техники (ICOT) в Токио. Обоснование проекта и его стратегическая линия излагаются как в общих терминах, так и достаточно детально в докладах Фучи (1981) и Мото-ока (1981), сделанных на конференции в JIPDEC. Объясняя, что техника обработки информации должна в ближайшем будущем охватить по существу всю деятельность в промышленности и управлении, в областях образования и культуры, они подчеркивают необходимость создания настолько совершенных вычислительных машин, чтобы с ними легко могли работать самые широкие круги пользователей. Такие машины должны обладать способностями к широкомасштабному приобретению знаний, самообучению, рассуждениям и решению задач, и их взаимодействие с человеком должно быть ориентировано на последнего. Главным образом именно этими требованиями обусловлена необходимость в значительном расширении возможностей обработки данных при помощи программного обеспечения очень высокого уровня.

Очевидно, что эти устремления потребуют радикальных изменений в образовании, положении и деятельности разработчиков программного обеспечения, программистов и пользователей.

Например, вследствие того что в умах традиционных программистов глубоко и прочно укоренился процедурный подход к программированию, у многих из них может возникнуть серьезное психологическое сопротивление принятию декларативных формальных систем. Еще одним возможным препятствием добровольным изменениям может быть то коммерческое обстоятельство, что на разработку традиционных аппаратных средств и программного обеспечения уже были выделены огромные капиталовложения. Как было отмечено д'Агапаевом (1982), в ответ на японский проект создания ЭВМ пятого поколения остальной мир должен сделать выбор: «нововведения или стабильность». В своем собственном очевидном выборе японцы сделали ставку не только на уверенность в том, что расходы, связанные с переходом к новому поколению вычислительной техники, будут в конце концов компенсированы за счет конкурентоспособности этой техники на мировом рынке,— более фундаментальную ставку они сделали на саму возможность создания этой новой техники.

Предварительные разработки проекта ЭВМ пятого поколения были представлены в материалах конференции JIPDEC (1981) и в последующих японских публикациях в виде довольно абстрактных концептуальных диаграмм, составные части которых и взаимосвязи между ними не так-то легко разгадать. В самых общих терминах можно представлять себе, что система пятого поколения включает *моделирующую систему программного обеспечения* (МСПО), помещенную между рядом человеческих прикладных систем (ЧПС) и *аппаратной системой машины* (АСМ). Для формулировки ЧПС предполагается использовать широкий спектр ориентированных на человека языков, включающий речь, естественные языки и графические изображения. ЧПС связаны с МСПО посредством интерфейса, обладающего способностями как понимать содержащиеся в ЧПС знания, так и представлять их в подходящей для интеллектуального синтеза программ и обработки этих знаний форме. В ядре МСПО находится интеллектуальная система программирования и целый ряд систем баз знаний, с помощью которых МСПО может строить обрабатываемые машиной представления как знаний, так и программ, необходимых для осуществления целей, поставленных ЧПС. Функция синтеза является главной особенностью интерфейса МСПО-АСМ. Сама аппаратная система машины включает машину решения задач и машину баз знаний, которые сами состоят из еще большего числа разнообразных машин, приспособленных для численных расчетов, манипулирования символами и работы с базами данных. Подробная интерпретация всех этих предложений дается в статье Треливана (1982).

Предполагаемая в проекте максимальная производительность вычислительных машин пятого поколения очень высока: обычно приводится число порядка 10^7 клс. Такое быстродействие будет достигаться за счет совместного использования до 10^4 параллельных процессоров, обращающихся к памяти емкостью до 10^4 мегабайт. Однако ближайшей целью проекта, рассчитанной на первый трехлетний период, является создание персональной рабочей станции лишь с одним процессором, дающим 20—30 клс и обращающимся к памяти емкостью 10—20 Мб. В терминах скорости построения выводов эта настольная машина будет достигать почти той же производительности, что и скомпилированный DEC-10 Пролог на вычислительной машине DEC-2060. В дальнейшем предполагается создать более мощную персональную рабочую станцию, называемую МПВ (*машина параллельного вывода*). Она будет обладать 32 параллельными процессорами, совместно дающими 10^6 — 10^7 клс. Эта машина станет грозным конкурентом для многих существующих традиционных больших компьютеров; как ожидается, она будет готова до 1990 года. Ее прототип под названием TOPSTAR-II, включающий 24 параллельных микропроцессоров Z80 и работающий с системой параллельного вывода, называемой Паралог, уже построен и испытан.

Сейчас интерес всего мира сосредоточен естественно на первой ожидаемой машине, которая известна как ПМПВ (*персональная машина последовательного вывода*). Ее внутренние характеристики, как они представляются в настоящее время, описаны в отчете ICOT Нисикавой и др. (1983). Главную роль в ней будет играть аппаратура центрального процессора, предназначенного для быстрого выполнения унификации и поиска связываний, а также для сборки мусора, управляемой программно-аппаратными средствами. Каждое из ее слов длиной 40 бит включает поле данных из 32 бит и, кроме того, тег для динамического контроля типов длиной 6 бит и двухбитное поле для управления сборкой мусора.

Кроме претенциозной программы, касающейся аппаратных средств, большое удивление после опубликования планов создания ЭВМ пятого поколения вызвал выбор в качестве основного формализма языка логического программирования. Этот выбор обосновывается несколькими факторами, которые отчетливо сформулированы в статье Фурукавы и др. (1981), содержащейся в материалах конференции в JIPDEC. Среди этих факторов отмечается то, что логика покрывает как функциональное программирование, так и формальные системы реляционных баз данных, и что вместе с тем она дает единообразное представление данных, программ, спецификаций и понятий метауровня. В своей более поздней оценке возможностей логики

Ковальский (1982) объясняет, что она обеспечивает важные связи между целями, стоящими перед техникой программного обеспечения и искусственным интеллектом, а также между основанным на правилах программированием и технологией СБИС.

В персональной машине последовательного вывода в качестве нулевой версии языка ядра (короче, ЯЯО) применяется модифицированный вариант DEC-10 Пролога, характерными особенностями которого являются обычное устройство управления, след, локальный и глобальный стеки, обеспечивающие обычную модель совместного использования структур. С точки зрения системного программирования ЯЯО будет играть почти такую же роль, что и язык ассемблера в традиционных машинах, но при этом обладать бесконечно большими возможностями. ЯЯО будет использован для написания большей части операционной системы ПМПВ и связанного с ней программного обеспечения. В то же время он будет служить в качестве языка высокого уровня для взаимодействия с пользователем и решения задач. Основанный на логике язык ядра машины параллельного вывода, известный как ЯЯ1, будет, по всей видимости, базироваться на языках Парлог и Concurrent Prolog. Интересный исторический взгляд, личные впечатления и анекдоты, связанные с проектом создания ЭВМ пятого поколения и ролью логики в нем, представлены Уорреном (1982) и Шапиро (1983а).

Японский проект создания систем ЭВМ пятого поколения стимулировал принятие финансируемых правительствами программ создания нового поколения вычислительной техники в Великобритании (см., например, Отчет комитета Альви, 1984), Европе и Соединенных Штатах. Он привел также к значительному возрастанию интереса к логическому программированию во всем мире. Насколько далеко будет простираться этот интерес, и выдержит ли он проверку временем, покажет будущее. Если работы по проекту создания ЭВМ пятого поколения приблизятся к достижению сформулированных в нем целей, то вызов, брошенный специалистам в области вычислительной техники всего остального мира, будет очень серьезным: как заявил Кацухиро Фучи на конференции в JIPDEC, вопрос будет стоять тогда так: «тоттаться на месте или идти вперед, ибо другого выбора нет».

Литература

Абрамсон (Abramson H.)

1983 Definite clause transition grammars. Research Report, Dept. of Computer Science, Univ. of British Columbia, Vancouver.

Андреака и Немети (Andreka H. and Nemeti I.)

1976 The generalized completeness of Horn predicate logic as a programming language. Research Report 21, Dept. Artificial Intelligence, Univ. of Edinburgh, Scotland.

Апт и ван Эмден (Apt K. R. and van Emden M. N.)

1982 Contributions to the theory of logic programming, Journal of the ACM 29(3), 842—862.

Атлантик Сити (Atlantic City)

1984 International Symposium on Logic Programming, Atlantic City, New Jersey, February 6—9. IEEE Computer Society Press, New York.

Балог (Balogh K.)

1981 On an interactive program verifier for PROLOG programs. См. Салгаторьян (1981).

Банди, Бирд, Люгер, Меллиш и Палмер (Bundy A., Byrd L., Luger G., Mellish C. S. and Palmer M.)

Solving mechanics problems using meta-level inference. Proc. of 6th Int. Joint Conf. on Artificial Intelligence, Tokyo, pp. 1017—1027.

Барстолл и Дарлингтон (Burstall R. M. and Darlington J.)

1977 A transformation system for developing recursive programs. Journal of the ACM 24(1), 44—67.

Баттани и Мелони (Battani G. and Meloni H.)

1973 Interpreteur du langage de programmation PROLOG. Research Report, Artificial Intelligence Group, Univ. of Aix—Marseille, Luminy, France.

Бендел, Ковс и Середи (Bendl J., Kovcs P. and Szeregi P.)

1980 The MPROLOG system. См. Галлер и Минкер (1978).

Бирд (Byrd L.)

1980 Understanding the control flow of PROLOG programs, Logic Programming Workshop (1980).

Бобров (Bobrow D. G.)

1980 (Editor). Special issue on non-monotonic logic, Artificial Intelligence 13.

Бойер и Мур (Boyer R. S. and Moore J. S.)

1972 The sharing of structure in theorem proving programs. In Machine Intelligence, Vol. 7, (B. Meltzer and D. Michie, eds.) 101—116. Edinburgh University Press, Scotland.

1977 A fast string searching algorithm. Comm. of the ACM 20 (10), 762—772.

Боргвард (Borgwardt P.)

1984 Parallel PROLOG using stack segments on shared memory multiprocessors. См. Атлантик Сити (1984).

Боуэн (Bowen K. A.)

1980 Programming with full first order logic. Research Report, School of

- Computer and Information Science (November). Syracuse University, New York.
- Боузин и Ковальский** (Bowen K. A. and Kowalski R. A.)
 1982 Amalgamating language and metalanguage in logic programming. См. Кларк и Териленд (1982).
- Браф и ван Эмден** (Brough D. R. and van Emden M. H.)
 Dataflow, flowcharts and LUCID-style programming in logic. См. Атлантик Сити (1984).
- Браунхе** (Bruynooghe M.)
 1976 An interpreter for predicate logic programs: Part I. Report CW, Applied Mathematics and Programming Division. Katholieke Universiteit, Leuven, Belgium.
- 1981 Intelligent Backtracking for an interpreter of Horn clause logic programs. См. Салготарьян (1981).
- 1982 The memory management of PROLOG implementations. См. Кларк и Териленд (1982).
- Бриггс** (Briggs, J.)
 1984 Designing and implementing a child orientated interface to micro-PROLOG. См. LP Research Reports.
- British Computer Society**
 1983 Proceeding of Conference on Expert Systems, British Computer Society, Churchill College, Univ. of Cambridge, December 14–16.
- Бэкус** (Basckus J.)
 1978 Can programming be liberated from the von Neumann style? ACM Turing Award Lecture, Comm. of the ACM, 21 (8), 613–641.
- ван Эмден** (van Emden)
 1977a Programming in resolution logic. «Machine Intelligence» Vol. 8 (E. W. Elcock and R. Michie, eds.), 266–299. Ellis Horwood Ltd., Chichester, England.
- 1977b Relational equations, grammars and programs. Proc. of Conf. on Theoretical Computer Science, Univ. of Waterloo, Ontario, Canada.
- 1978 Computation and deductive information retrieval. In «Formal Description of Programming Concepts» (E. Neuhold, ed.), 421–440. North Holland Publ., Amsterdam).
- ван Эмден и де Люсена Фило** (van Emden M. H. and de Lucena Filho G. J.)
 1982 Predicate logic as language for parallel programming. См. Кларк и Териленд (1982).
- ван Эмден и Ковальский** (van Emden M. H. and Kowalski R. A.)
 1976 The semantics of predicate logic as a programming language. Journal of the ACM 23(4), 733–742.
- Габбай и Сергот** (Gabbay D. M. and Sergot M. J.)
 1984 Negation as inconsistency. DOC Report 84/7. См. DOC Reports.
- Галлер** (Gallaire H.)
 1981 The impact of logic on databases. Cannes, France.
- Галлер, Лассер** (Gallaire H. and Lassere C.)
 1982 Metalevel control for logic programming. См. Кларк и Териленд (1982).
- Галлер, Минкер** (Gallaire H. and Minker J.)
 1978 (Editors). «Logic and Data Bases». Plenum Press, New York.
- Грин** (Green C. C.)
 1969 The application of theorem proving to problem solving. Proc. of 1st Int. Joint Conf. on Artificial Intelligence, Washington, D. C., pp. 219–240.
- Гард и Уотсон** (Gurd J. and Watson I.)
 1980 A multilayered dataflow computer architecture, Research Report, Dept. of Computer Science, Univ. of Manchester, England.
- д'Аранаев** (d'Agapayeff A.)

- 1982 An Introduction to the fifth generation. См. SPL (1982).
Дал (Dahl V.)
 1980 Two solutions for the negation problem. См. Logic Programming Workshop (1980).
 1981 On data base system development through logic. Research Report, Dept. of Mathematics, Faculty of Exact Sciences, Univ. of Buenos Aires, Argentina.
- Дарлингтон и Ковальский (Darlington J. and Kowalski R. A.)**
 1983 (Editors). Declarative systems architecture. SERC-DOI IKBS Architecture Study, Vol. 2, U. K. Government Department of Trade and Industry, London.
- Дарлингтон и Рив (Darlington J. and Reeve M.)**
 1981 ALICE; a multi-processor reduction machine for the parallel evaluation of applicative languages. Proc. of ACM Conf. of Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire.
 1983 ALICE and the parallel evaluation of logic programs. Invited Paper to 19th Annual Int. Symposium on Computer Architecture, Stockholm, Sweden.
- Делиянин, Ковальский (Deliyanni A. and Kowalski R. A.)**
 1979 Logic and semantic networks. Comm. of the ACM 22(3), 184—192.
- Де Лонг (De Long H.)**
 1970 «A profile of Mathematical Logic.» Addison — Wesley, Reading, Massachusetts.
- Дейкстра (Dijkstra, E. W.)**
 1976 «A Discipline of Programming», Prentice-Hall, Englewood Cliffs, New Jersey. Русский перевод: Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.
- DOC Reports**
 1975 (Dept. of Computing Reports, Imperial College of Science of and Technology, London, 1984 England).
- Дискуссия комиссии Альви (Alvey Debate)**
 1984 Debate on Information Technology: The Alvey Report. House of Lords, HANSARD, January 18th.
- Джаффар, Лассез и Ллойд (Jaffar J., Lassez J. L. and Lloyd J. W.)**
 1983 Completeness of the negation as failure rule. Proc. of 8th Int. Joint Conf. on Artificial Intelligence, Karlsruhe, Germany.
- JIPDEC**
 1981 Proc. of Int. Conf. on Fifth Generation Computer Systems, Japan Information Processing Development Centre, Tokyo. Republished 1982. (T. Moto-oka, ed.). North Holland Publ., Amsterdam.
- Дэвис (Davis M.)**
 1958 «Computability and Unsolvability», McGraw-Hill, New York.
- Злуф (Zloof M. M.)**
 1975 Query-by-Example. «Proceedings of AFIPS-75 National Computer Conference», Vol. 4, AFIPS Press, Montvale, New Jersey.
- Кларк (Clark K. L.)**
 1977 The synthesis and verification of logic programs. См. LP Research Reports.
 1978 Negation as failure. См. Галлер и Минкер (1978).
 1979 Predicate logic as a computational formalism. Ph. D. Thesis. Imperial College of Science and Technology, Univ. of London, England.
- Кларк и Дарлингтон (Clark K. L. and Darlington J.)**
 1980 Algorithm classification through synthesis. The Computer Journal 23(1), 61—65.
- Кларк и Грегори (Clark K. L. and Gregory S.)**
 1981 A relational language for parallel programming. Proc. of ACM Conf.

- on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire.
- 1983 PARLOG: a parallel logic programming language. Report DOC 83/5. См. DOC Reports.
- 1984 PARLOG: parallel programming in logic. Report Doc 84/4. См. DOC Reports.
- Кларк и Ковальский (Clark K. L. and Kowalski R. A.)
- 1977 Predicate logic as programming language. См. LP Research Reports.
- Кларк и Маккеиб (Clark K. L. and McCabe F. G.)
- 1979a Programmers' guide to IC-PROLOG. Report DOC 79/7. См. DOC Reports.
- 1979b The control facilities of IC-PROLOG. См. Мичи (1979).
- 1980 IC-PROLOG: aspects of its implementation. См. Logic Programming Workshop (1980).
- 1982 PROLOG: a language for implementing expert systems. In «Machine Intelligence», Vol. 10 (J. E. Hayes, D. Michie, and Y. H. Pao, eds.), pp. 455—470. Ellis Horwood Ltd., Chichester, England.
- 1984 «Micro-PROLOG: Programming in Logic», Prentice Hall, Englewood Cliffs, New Jersey. Русский перевод: Кларк К., Маккеиб Ф. Введение в логическое программирование на микро-Прологе.—М.: Радио и Связь, 1987.
- Кларк и Сикель (Clark K. L. and Sickel S.)
- 1977 Predicate logic: a calculus for deriving programs. Proc. of 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts.
- Кларк и Терилиуд (Clark K. L. and Tarnlund S.-A.)
- 1977 A first order theory of data and programs. Proc. of IFIP-77, Toronto, pp. 939—944. North Holland Publ., Amsterdam.
- 1982 (Editors). «Logic Programming» (APIC Studies in Data Processing, Vol. 16). Academic Press, London.
- Кларк и ван Эмден (Clark K. L. and van Emden M. H.)
- 1981 Consequence verification of flowcharts. IEEE Transactions on Software Engineering SE-7(1), 52—60.
- Кларк, Маккеиб и Грегори (Clark K. L., McCabe F. G. and Gregory S.)
- 1982 IC-Prolog language features. См. Кларк и Терилиуд (1982).
- Кларк, Маккиман и Сикель (Clark K. L., McKeeman W. and Sickel S.)
- 1982 Logic program specification of numerical integration. См. Кларк и Терилиуд (1982).
- Клини (Kleene S.)
- 1952 «Introduction to Metamathematics». Van Nostrand, New York. Русский перевод: Клини С. К. Введение в метаматематику. М.: ИЛ, 1957.
- Клоксин и Меллиш (Clocksin W. F. and Melish C. S.)
- 1980 The UNIX PROLOG system. Software Report 5, Dept. of Artificial Intelligence, Uni of Edinburgh, Scotland.
- 1981 «Programming in PROLOG», Springer-Verlag, Berlin. Русский перевод: Клоксин У., Меллиш К. Программирование на языке Пролог.—М.: Мир, 1987.
- Клужняк и Шпакович (Kluzniak F. and Szpakowicz S.)
- 1982 PROLOG for programmers: and outline of a teaching method. Logic Programming Newsletter No. 3 (L. M. Pereira, ed.). Univ. Nova de Lisbon, Portugal.
- Кнут, Моррис и Пратт (Knuth, D. E., Morris, J. H. and Pratt, V. R.)
- 1976 Fast pattern matching in strings. SIAM Journal of Computing 5, 90—99.
- Ковальский (Kowalski, R. A.)
- 1974a Logic for problem solving. DCL Memo No. 75, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland.

- 1974 Predicate logic as a programming language. Proc. of IFIP-74. North Holland Publ., Amsterdam, pp. 569—574.
- 1975 A proof procedure using connection graphs. Journal of the ACM, 22(4), 572—595.
- 1978 Logic for data description. См. Галлер и Минкер (1978).
- 1979a «Logic for problem Solving» (Artificial Intelligence Series, Vol. 7). Elsevier — North-Holland, New York.
- 1979 Algorithm = logic + control. Comm. of the ACM 22, 424—431.
- 1981a Logic as a database language. Proc. of Workshop on Database Theory, Cetraro, Italy.
- 1981b PROLOG as a logic programming language. Proc. of AICA Congress, Pavia, Italy, September 23—25.
- 1982 Logic Programming in the fifth generation. См. SPL (1982).
- 1983a The relationship between logic programming and logic specification. Invited Paper to BCS-FACS/SERC Workshop on Program Specification and Verification, Univ. of York, March 28—30.
- 1983b Logic Programming, Invited Paper to IFIP-83, Paris, France.
- 1983c The frame problem in logic databases. См. LP Research Reports.
- 1983d Logic for expert systems. См. British Computer Society (1983).
- 1984 The History of logic programming. См. LP Research Reports.
- Ковалевский и Кюнер (Kowalski R. A. and Kuehner D. G.)**
- 1971 Linear resolution with selector function. Artificial Intelligence 2, 227—260.
- Колмероэ (Colmerauer A.)**
- 1978 Metamorphosis grammars. In «Natural Language Communication with Computers» (L. Boic, ed.) (Lecture Notes on Computer Science, No. 63) pp. 133—189. Springer-Verlag, Berlin.
- 1981 An interesting subset of natural language. См. Кларк и Тернлинд (1982).
- Колмероэ, Кануи, Пазеро, Руссель (Colmerauer A., Kanoui H., Pasero, R. and Roussel P.)**
- 1973 Un system de communication homme — machine en Francais. Research Report, Artificial Intelligence Group. Univ. of Aix — Marseille, Luminy, France.
- Конери (Conery J. S.)**
- 1983 The AND/OR Process model for parallel interpretation of logic programs. Technical Report 204 (Ph. D. Thesis) (June). Univ. of California at Irvine.
- Конери и Кильдер (Conery J. S. and Kibler D. F.)**
- 1981 Parallel interpretation of logic programs. Proc. of ACM Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire, pp. 163—170.
- Кори, Хаммонд, Ковалевский, Кривачек, Садри и Сергот (Cory, H. T., Hammond P., Kowalski R. A., Kriwaczek F., Sadri F. and Sergot M. J.)**
- 1984 The British Nationality Act as a logic program. См. Research Reports.
- Квайн (Quine W. V. O.)**
- 1959 «Methods of Logic» (2nd Ed.). Routledge and Kegan Paul, London.
- Лассез и Марр (Lassez J. L. and Maher M. J.)**
- 1983 Optimal fixedpoints of logic programs. Research Report, Dept. of Computer Science. Univ. of Melbourne, Australia.
- Лихтман (Lichtman B. M.)**
- 1975 Features of very High-level programming with PROLOG. M. Sc. Thesis, Dept. of Computing, Imperial College, Univ. of London.
- Ллойд (Lloyd J. W.)**
- 1983a Mathematical Foundations of logic programming. Research Mono-

- graphi, Dept. of Computer Science (May). Univ. of Melbourne, Australia.
- 1983 An introduction to deductive database systems. The Australian Computer Journal 15(2), 52—57.
- Logic Programming Workshop**
- 1980 Proc. of Int. Workshop on Logic Programming, von Neumann Comp. Sci. Soc., Debrecen, Hungary, July 14—16.
- Logic Programming Workshop**
- 1983 Proc. of Int. Workshop on Logic Programming, Nucleo de Inteligencia Artificial, Univ. Nova de Lisboa, Albufeira, Algarve, Portugal, June 29 — July 1.
- LP Research Reports**
- 1975 Logic Programming Research Reports, Theory of Computing.
- 1984 Research Group, Dept. of Computing. Imperial College of Science and Technology, London, England.
- Манна (Manna Z.)**
- 1974 «Mathematical Theory of Computation». McGraw-Hill, New York.
- Маккиман и Сикель (McKeeman W. and Sickel S.)**
- 1980 Hoare's program FIND revisited. См. Logic Programming Workshop (1980).
- Манна и Уолдингер (Manna Z. and Waldinger R.)**
- 1980 A deductive approach to programm synthesis. ACM Transactions on Programming Language and Systems 2(1), 90—121.
- Маррэй (Murray N.)**
- 1978 A proof procedure for non-clausal first order logic. Research Report, School of Computer and Information Science. Syracuse University, New York.
- Марсель (Marseille)**
- 1982 Proc. of 1st Int. Logic Programming Conference, Faculté des Sciences de Luminy, Marseille, France, Septembre 14—17.
- Меллиш (Mellish C. S.)**
- 1982 An alternative to structure sharing in the implementation of a PROLOG interpreter. См. Кларк и Тернлунд (1982).
- Минкер (Minker J.)**
- 1981 On indefinite databases and the closed world assumption. Research Report, Dept. of Computer Science. Univ. of Maryland, College Park, Maryland.
- 1983 AI and database research laboratory at the University of Maryland. Logic Programming Newsletter No. 5 (L.M. Pereira, ed.). Univ. Nova de Lisboa, Lisbon, Portugal.
- Минский (Minsky, M. L.)**
- 1975 A framework for the representation of knowledge. In «The Psychology of Computer Vision» (P. Winston, ed.), pp. 211—280. McGraw-Hill, New York.
- Мичи (Michie, D.)**
- 1979 (Editor). «Expert Systems in the Microelectronic Age». Edinburgh University Press, Scotland.
- Мото-ока (Moto-oka, T.)**
- 1981 Challenge for knowledge information processing systems. См. JIPDEC (1981).
- Нивс, Бэкхause, Андерсон, Уильямс (Neves J. C., Backhouse R. C., Anderson S. O. and Williams M. H.)**
- 1982 A PROLOG implementation of Query by Example. Research Report, Heriot) — Watt University, Edinburgh, Scotland.
- Никола, Галлер (Nicolas J. M. and Gallaire H.)**
- 1978 Database: theory versus interpretation. См. Галлер и Минкер (1978).

- Нильсон (Nilsson N. J.)**
- 1971 «Problem-Solving Methods in Artificial Intelligence». McGraw-Hill, New York. Русский перевод: Нильсон Н. Искусственный интеллект. Методы поиска решения. — М.: Мир, 1973.
- Нисикава, Екота, Ямamoto, Таки и Утида (Nishikawa H., Yokota M., Yamamoto A., Taki K. and Uchida S.)**
- 1983 The personal inference machine (PSI): its design philosophy and machine architecture. ICOT Technical Report TR-013. Institute for New Generation Computing Technology, Tokyo, Japan.
- Отчет комитета Альви (Alvey Report)**
- 1982 A programme for advanced information technology. The report of the Alvey Committee, HMSO, London.
- Перейра (Pereira F. C. N.)**
- 1983 Logic for natural language analysis. Research Report, Artificial Intelligence Centre, Computer Science & Technology Division, SRI International, Menlo Park, California.
- Перейра и Уоррен (Pereira F. C. N. and Warren D. H. D.)**
- 1980 Definite clause grammars for language analysis — a survey on the formalism and a comparison with augmented transition networks. Artificial Intelligence 13, 231—278.
- Перейра (Pereira L. M.)**
- 1982 Logic control with logic. См. Марсель (1982).
- Перейра и Монтеиро (Pereira L. M. and Monteiro L. F.)**
- 1981 The semantics of parallelism and corouting in logic programming. См. Салготарьян (1981).
- Перейра и Порто (Pereira L. M. and Porto A.)**
- 1982 Selective Backtracking. См. Кларк и Терилуид (1982).
- Перейра, Сабатье и Оливейра (Pereira L. M., Sabatier O. and Oliveira E.)**
- 1982 Orbi: an expert system for environment resource evaluation through natural language. См. Марсель (1982).
- Поллард (Pollard G. H.)**
- 1981 Parallel execution of Horn clause programs. HP. Thesis. Imperial College of Science and Technology, Univ. of London, England.
- Рафаэл (Raphael, B.)**
- 1971 The frame problem in problem solving systems. Proc. of Advanced Study Institute on Artificial Intelligence and Heuristic Programming, 1970. Menaggio, Italy. Republished (N. V. Findler and B. Meltzer, eds.), pp. 159—169. Edinburgh University Press, Scotland.
- Робертс (Roberts, G. M.)**
- 1977 An implementation of PROLOG. M. Sc. Thesis, Univ. of Waterloo, Ontario, Canada.
- Робинсон (Robinson J. A.)**
- 1965 A machine-oriented logic based on the resolution principle Journal of the ACM 12, 23—41. Русский перевод: Машинно-ориентированная логика, основанная на принципе резолюции. — Кибери. сб., нов. сер., 7. М.: Мир, 1970.
- 1979 «Logic: form and functions». Edinburgh University Press, Scotland, and Elsevier — North Holland, New York.
- Робинсон и Сиберт (Robinson J. A. and Sibert E. E.)**
- 1980 Logic programming in LISP. Research Report, School of Computer and Information Science. Syracuse University, New York.
- Руссель (Roussel, P.).**
- 1975 PROLOG: manuel de reference et d'utilisation. Research Report, Artificial Intelligence Group. Univ. of Aix — Marseille, Luminy, France.
- Салготарьян (Salgotarjan)**
- 1981 Proc. of Colloquium on Mathematical Logic Programming, 1978, Salgotarjan, Hungary. Republished (B. Domoki and T. Gergely, eds.). North-Holland Publ., Amsterdam.

- Саммут и Саммут** (Sammut R. A. and Sammut C. A.)
 1983a PROLOG: a tutorial introduction. Australian Computer Journal 15, 42—51.
 1983b The implementation of UNSW-PROLOG. Australian Computer Journal 15, 58—64.
- Себелик и Степанек** (Sobelik J. and Stepanek, P.)
 1980 Horn clause programs suggested by recursive functions. См., Logic Programming Workshop (1980).
- Сергот** (Sergot, M. J.)
 1982 Prospects for representing the law as logic programs. См. Кларк и Тернлунд (1982).
 1983a Logic databases and state transitions. Proc. of Workshop on Uses of Database for Knowledge Bases, Univ. of Aberdeen Scotland, April 14th.
 1983 A Query-the-User facility for logic programming. In «Integrated Interactive Computer Systems» (P. Degano and E. Sandewall, eds.). North-Holland Publ., Amsterdam.
- Середи** (Szeredi P.)
 1981 Mixed language programming — a method for producing efficient PROLOG programs. Proc. of Workshop on Logic Programming for Intelligent Systems, Los Angeles, California, August.
- Слоуман** (Sloman, A.)
 1983 Intelligent systems: a brief overview. SERC-DOI IKBS Architecture Study, Vol. 1 Annexes, U. K. Government of Trade and Industry, London.
 1982 Proc. of Int. Conf. on The Fifth Generation: Dawn of the Second Computer Age. SPL International, London, July 7—9.
- Стивенс** (Stevens C.)
 1977 The application of call-by need to automatic theorem proving. M. Sc. Thesis, Dept. of Computing Imperial College, Univ. of London, England.
- Сэндуолл** (Sandewall E.)
 1973 Conversion of predicate-calculus axioms, viewed as nondeterministic programs. Proc. of 3rd Int. Joint Conf. on Artificial Intelligence, Stanford, California, pp. 230—234.
- Тарский** (Tarski A.)
 1969 Truth and proof. Scientific American 220(6), 63—77.
- Тернлунд** (Tarnlund, S. A.)
 1975a An interpreter for the programming language predicate logic. Proc. of 4th Int. Joint Conf. on Artificial Intelligence, Tbilisi, Georgia, USSR, pp. 601—608.
 1975b Logic information processing. Report TRITA-IBADB 1034, Dept. of Information Processing and Computer Science. The Royal Institute of Technology and Univ. of Stockholm, Sweden.
 1977 Horn clause computability, BIT 17, 215—226.
 1978 An axiomatic data base theory. См. Галлер и Минкер (1978).
- Тик и Уоррен** (Tick E. and Warren D. H. D.)
 1984 Towards a pipelined PROLOG processor. См. Atlantic City (1984).
- Трелиаван** (Treleavan P.)
 1982 Japan's fifth generation computer systems project. См. SPL (1984).
- Уинтерстайн, Даусман и Перш** (Winterstein G., Dausman M. and Persch G.)
 1980 Deriving different unification algorithms from a specification in logic. См. Logic Programming Workshop (1980).
- Уоллис** (Wallis P. J. L.)
 1982 (Editor). «Programming Technology: State of the Art Report. Pergamon Infotech Ltd., Maidenhead, England.
- Уоррен** (Warren D. H. D.)

- 1977a Implementing PROLOG compiling predicate logic programs. Research Reports Nos. 39 and 40, Dept. of Artificial Intelligence. Univ. of Edinburgh, Scotland.
- 1977 Logic programming and compiler writing. Research Report 44, Dept. of Artificial Intelligence, Univ. of Edinburgh, Scotland.
- 1979 PROLOG on the DEC System-10. См. Мици (1979).
- 1980 An improved PROLOG implementation which optimizes tail recursion. См. Logic Programming Workshop (1980).
- 1981 Efficient processing of interactive relational database queries expressed in logic. Proc. of 7th Int. Conf. on Very Large Data Bases, Cannes, France.
- 1982 A view of the fifth generation and its impact. Proc. of Conf. on Japan and the Fifth Generation, Pergamon Infotech State of the Art Conference, London, September 27—29.
- Уоррен и ван Канегэм (Warren D. H. H. and van Canaghem M.)
- 1983 Logic programming and its applications. Ablex Publ., New Jersey.
- Уоррен, Переира и Переира (Warren D. H. D., Pereira L. M. and Pereira F. C. N.)
- 1977 PROLOG — the language and its implementation compared with LISP. Proc. of Symposium on Artificial Intelligence and Programming Languages, SIGPLAN Notices, Vo. 12, No. 8.
- 1979 User's guide to DEC System-10 PROLOG. Occasional Paper 15, Dept. of Artificial Intelligence. Univ. of Edinburgh, Scotland.
- Уппсала (Uppsala)
- 1984 Proc 2nd Int. Logic Programming Conference, Univ. of Uppsala, Sweden, July 2—7.
- Уэйрауч (Weyrauch R.)
- 1980 Prolegomena to a theory of mechanized formal reasoning. Artificial Intelligence 13, 133—170.
- Уэир (Weir D. J.)
- 1982 Teaching logic programming: an interactive approach. M. Sc. Thesis, Dept. of Computing. Imperial College, Univ. of London, England.
- Уелхем (Welham, R.)
- 1976 Geometry problem solving. Research Report 14, Dept. of Artificial Intelligence. Univ. of Edinburgh, Scotland.
- Фучи (Fuchi K.)
- 1981 Aiming for knowledge information processing. См. JIPDEC (1981).
- Фурукава, Накадзима, Енедзава, Гото и Аояма (Furukawa K., Nakajima R., Yonezawa A., Goto S. and Aoyama A.)
- 1981 Problem solving and inference mechanisms. См. JIPDEC (1981).
- Хаммонд (Hammond P.)
- 1980 Logic programming for expert systems. M. Sc. Thesis, Dept. of Computing. Imperial College, Univ. of London, England.
- 1983a APES: a user manual. Doc Report 82/9. См. DOC Reports.
- 1983b Representation of DHSS regulations as a logic program. См. British Computer Society (1983).
- Хаммонд и Сергот (Hammond R. and Sergot M. J.)
- 1983 A PROLOG shell for logic based expert systems. См. British Computer Society (1983).
- Ханссон и Йоханссон (Hansson B. and Johansson A. L.)
- 1980 Development of software for deductive reasoning. См. Logic Programming Workshop (1980).
- Ханссон и Тарнлунд (Hansson A. and Tarnlund S. A.)
- 1979 A natural programming calculus. Proc. of 6th Int. Joint Conf. on Artificial Intelligence, Tokyo.
- Хастлер (Hustler A.)

- 1982 Programming law in logic. Research Report, Dept. of Computer Science, Univ. of Waterloo, Ontario, Canada.
- Хейс (Hayes P. J.)**
- 1973 Computation and deduction: Proc. 2nd Symposium on Mathematical Foundations of Computer Science, Czechoslovak Academy of Sciences, pp. 105—108.
- Хеллерстайн и Шапиро (Hellerstein, L., and Shapiro, E. Y.)**
- 1984 The MAXFLOW experience. См. Атлантик Сити (1984).
- Хилл (Hill, R.)**
- 1974 LUSH resolution and its completeness. DSL Memo No. 78, Dept. of Artificial Intelligence. Univ. of Edinburgh, Scotland.
- Хоар (Hoare, C. A. R.)**
- 1969 Axiomatic basis for computer programming. Comm. of the ACM 12, 576—580.
- Ходжес (Hodges, W.)**
- 1977 «Logic». Penguin, Middlesex, England.
- Хоггер (Hogger, C. J.)**
- 1975 Stepwise refinement for the synthesis of predicate logic programs. См. LP Research Reports.
- 1976 A logic program for the linear programming Simplex algorithm. См. LP Research Reports.
- 1977 Deductive synthesis of logic programs. См. LP Research Reports.
- 1978a Program synthesis in predicate logic. Proc. of AISB/GI Conf. on Artificial Intelligence, Hamburg, Germany, July 18—20.
- 1978b Goal oriented derivation of logic programs. Proc. 7th Symposium on Mathematical Foundations of Computer Science, Polish Academy of Sciences, Zakopane, Poland.
- 1979a Derivation of logic programs. Ph. D. Thesis. Imperial College of Science and Technology, Univ. of London, England.
- 1979b Logic analysis of some string-matching algorithms. См. LP Research Reports.
- 1981 Derivation of logic programs. Journal of the ACM 28(2), 372—422.
- 1982a Logic programming and program verification. Invited Paper to Pergamon Infotech State of the Art Conference on Programming: New Directions, World Trade Centre, London, June 15—17. Republished in Wallis (1982).
- 1982b Concurrent logic programming. См. Кларк и Терилунд (1982).
- Чакраварти, Минкер, Трен (Chakravarthy U. S., Minker J. and Tran D.)**
- 1982 Interfacing predicate logic languages and relational databases. См. Марсель (1982).
- Чень и Ли (Chang C. L. and Lee R. C. T.)**
- 1973 «Symbolic Logic and Mechanical Theorem Proving», Academic Press, New York. Русский перевод: Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. — М.: Наука, 1983.
- Шантане-Тот, Середи (Santane-Toth E. and Szeredi P.)**
- PROLOG applications in Hungary. См. Кларк и Терилунд (1982).
- Шапиро (Shapiro E. Y.)**
- 1983a Japan's fifth generation computers project—a trip report, Report No. CS 83—07, Dept. of Applied Mathematics. Weizmann Institute of Science, Rehovot, Israel.
- 1983b A subset of Concurrent Prolog and its Interpreter. ICOT Technical Report TR-003, Institute for New Generation Computing Technology, Tokyo, Japan.
- Элкок (Elcock E. W.)**
- 1981 Logic and programming methodology, Departmental Report No. 80, Dept. of Computer Science, Univ. of Western Ontario, London, Ontario, Canada. Ontario, Canada.

- Элсон (Elson M.)
1973 «Concepts of Programming languages (Computer Science Series)»
Science Research Associates, Chicago.
- Энналс (Ennals J. R.)
1980 Logic as a computer language for children. См. LP Research Reports.
1981 History and computing. Collection of Papers 1979—1981, Report DOC
81/22 DOC Reports.
1982 Teaching logic as a computer language in schools. См. Марсель
(1982).
1984 «Beginning Micro-PROLOG» (2nd Ed.). Ellis Horwood Ltd., Chichester,
England and Heinemann Computer in Education Ltd., London,
England.
- Эрбран (Herbrand, J.)
1967 Investigations in proof theory. In «From Frege to Gödel» (J. Heijenoort, ed.), pp. 525—581. Harvard Univ. Press, Cambridge, Massachusetts.

Предметный указатель

Автоматическое доказательство теорем (automatic theorem proving) 47
Аксиома фрейма (frame axiom) 296
Активация вызова (call activation) 58
Активная процедура (active procedure) 230
Алгоритм Маркова (Markov algorithm) 280
Алгоритм поиска (search algorithm) 56, 68—69
Алгоритм унификации (unification algorithm) 270—273
Амальгамирование (amalgamation) 294—295, 298, 305
Антецедент (antecedent) 32
Аргумент (argument) 19

База данных (database) 144—145, 304
Базисная процедура (base procedure) 86

Ватерлоо-Пролог (Waterloo-PROLOG) 275
Верификация (verification) см. Правильность
Ветвление (branching) 77, 80—82
Возврат (backtracking) 70
— после неудачи (after failure) 229
— после успеха (after success) 230
Восстановление пространства стека (reclaiming stack space) 252—264
Встроенная процедура (built-in procedure) 88
— функция (built-in function) 89
Входной массив данных (input heap) 227
Выбор вызова (call selection) 61
— процедуры (procedure selection) 62—63
Вывод (derivation) 40—41
— процедур (procedure derivation) 193—196, 205—208, 219, 222

— сверху вниз (top-down derivation) 40
— успешный (successful derivation) 40
Вызов (call) 57—58
— латентный (latent call) 75
— отрицательный (negated call) 121—123
— процедуры (procedure call) 58
— решенный (solved call) 60
Высказывание (proposition) 19
Выход (из процедуры) (exit)
— неудачный (failure exit) 229
— успешный (successful exit) 229
Вычисление (computation) 56
— бесконечное (infinite computation) 64, 67, 73, 183—184
— завершенное (terminated computation) 64
— конечное (finite computation) 64
— незавершенное (unterminated computation) 64
— тупиковое (unsuccessful computation) 64
— успешное (successful computation) 63
Вычислимая функция (computable function) 278—280
Вычислимое отношение (computable relation) 278
Вычислимость по Тьюрингу (Turing-computability) 279
— отношение (computed relation) 168—169
Вычисляемое решение (computed solution) 168

Грамматика определенных днзыонков (definite clause grammar) 313
Граница поиска (search frontier) 70, 211—212
Граф связей (connection graph) 91

- Декларативный язык (declarative language) 301—303, 316
 Декларация вида (mode declaration) 270, 323
 Денотат (denotation) 281
 Дерево вычислений (computation tree) 65
 — поиска (search tree) 69
 Деструктивное присваивание (destructive assignment) 162, 264, 297, 299—300
 Диэзюнкт (clause) 49, 52, 286, 289
 — определенный (definite clause) 287
 Доказательство от противного (proof by contradiction) 33
 Допущение замкнутости мира (closed-world assumption) 121, 289—290
- Завершаемость (termination) 186—188
 Заголовок процедуры (procedure heading) 58
 Замыкание (множества процедур) (completion (of procedure set)) 290
 Запись активации (activation record), см. Фрейм
 Запрос (query) 13
 Знак пунктуации (punctuation symbol) 19
- Извлечение ответа (answer extraction) 42—44
 ИЛИ-параллелизм (OR-parallelism) 320—322
 Именование (naming) 16—17, 158—163
 Имитация процесса возврата (simulation of backtracking) 116—118
 Импликация (implication) 32
 Имя процедуры (procedure name) 58
 Инвертируемость (invertibility) 80
 Индексирование (indexing) 146—148, 228, 236, 273
 Индивидуум (individual) 16—17, 27
 Интеллектуальная система, основанная на знаниях (intelligent knowledge-based system) 304—306
 Интеллектуальный возврат (intelligent backtracking) 274
 Интенциональное представление (intensional representation) 144
 Интервал целевого утверждения (span of goal) 167
- Интерпретатор (interpreter) 12, 55—56, 227
 Интерпретация (interpretation) 26—30, 282—283
 — процедурная (procedure interpretation) 57—63
 Инфиксная запись (infix notation) 23
 И-параллелизм (AND-parallelism) 320—322
 Искусственный интеллект (artificial intelligence) 47, 301, 304—306
 Исполнение (execution), см. также Стек глобальный, Стек локальный, Стек копий
 — логической программы (of logic program) 55—56
 — в сопрограммном режиме (cogouting) 99, 142—143
 — в сопрограммном режиме под управлением потока данных (dataflow coroutining) 142—143, 317—318
 — под управлением потока данных (dataflow execution) 318—319
 Истинностное значение (truth value) 28
 Итерация (iteration) 82—86
- Кандидат (для вызова) (candidate procedure) 228—229
 Квазинегация (quasi-negation) см. Отрицание как неудача
 Квантификация (quantification) 21
 Квантор всеобщности (universal quantifier) 22
 — существования (existential quantifier) 22
 Компиляция (compilation) 269
 Конечное пространство вычислений (finite computation space) 190
 Консеквент (consequent) 32
 Контекстно-зависимая грамматика (context-dependent grammar) 313
 Контекстно-свободная грамматика (context-free grammar) 312
 Контроль (соответствия) типов (type checking)
 — множеств фактов (of assertion sets) 145—146
 — термов (of terms) 142—143
 Корректность резолюции (soundness of resolution) 47, 286
- Лисп (LISP) 132, 164, 288
 Литера (literal) 52, 286

- Логика первого порядка (first-order logic) 16, 22
 — предикатов (predicate logic) см. *Логика первого порядка*
 Логическая и управляющая компоненты (logic and control components) 95, 128—129, 135
 Логическая программа (logic program) 16
 — формулировка баз данных (logic formulation of databases) 306—309
 — эквивалентность (logical equivalence) 43
 Логический алгоритм (logic algorithm) 95
 — вывод (logical inference) 16, 32—35
 Логическое следование (logical implication) 15, 26, 30—31
 Логлисп (LOGLISP) 288
- Макрообработка (macroprocessing) 222
 Марсельский Пролог (Marseille PROLOG) 91, 151, 273—275
 Машина параллельного вывода (Parallel Inference Machine) 326
 — потока данных (dataflow machine) 316
 — фон Неймана (von Neumann machine) 299, 315
 Метаморфозная грамматика (metamorphosis grammar) 313
 Метаязык (metalanguage) 293
 Методология программирования (programming methodology) 298—304
 Микро-Пролог (micro-PROLOG) 276, 293, 311, 314—315
 Миоключевое хэширование (multi-key hashing) 309
 Множество процедур (procedure set) 54
 — фактов (assertion set) 132
 — как входные и выходные данные (as input—output) 150—158
 — — массив (as array) 148—149
 — — структура данных (as data structure) 132, 143—163
 — — фильтро-неудачных вызовов (finite failure set) 290
 Моделирование поведения снизу вверх (bottom-up simulation) 102—110, 156—157
 Модель (model) 46
 — И/ИЛИ процессов (AND/OR process model) 322
- Молекула (molecule) 249
 МПролог (MPROLOG) 274
- Недетерминизм (non-determinism) 68, 74, 110—118
 Неподвижная точка (fixpoint) 284
 — — наибольшая (greatest fixpoint) 285
 — — наименьшая (least fixpoint) 284
 — — оптимальная (optimal fixpoint) 286
 Неразрешимая программа (unsolvable program) 67, 171, 286
 Неразрешимость (unsolvability) 180
 Нерезолютивный вывод (non-resolution inference) 213
 Нестандартная логика (non-standard logic) 280
- Область интерпретации (domain of discourse) 27
 Обработка естественного языка (natural language processing) 304, 312—313
 Обучение (education) 313—315
 Общая структура логической программы (general structure of logic program) 54—55
 Общезначимость (validity) 279, 285
 Общий пример (common instance) 36—37
 Объектный язык (object language) 293
 Ограничение целостности (integrity constraint) 146, 308
 Оператор отсечения (cut operation) 72
 Операция вызова процедуры (procedure calling operation) 58—60, 235—237
 Оптимизация последнего вызова (last-call optimization) 259—265
 Определяемое (definiandum) 192
 Определяющее (definiens) 192
 Оправдание (refutation) 42
 Основной (ground) 22
 — пример дизъюнкта (ground clause instance) 283
 Отношение (relation) 17—19
 — доказуемости (provability relation) 294
 — тождества (identity relation) 25
 Отрицание (формула) (denial) 32
 — исходное (initial denial) 40
 — пустое (empty denial) 33
 Отрицание (связка) (negation) 118—123, 288—292

- как неудача (*negation-as-failure*) 121—123, 130, 288—292
- как противоречие (*negation-as-inconsistency*) 292
- классическое (*classical*) 290—292
- Охрана (*guard*) 323

- Параллелизм (parallelism)** 303, 315—316, 319—324
- конвейерный (*pipelined parallelism*) 322
- Параллельный Пролог (Concurrent Prolog)** 323, 327
- процесс (*concurrent process*) 320
- Паралог (PARALOG)** 326
- Парлог (PARLOG)** 323, 327
- Переименование переменных (*renaming of variables*) 39, 44
- Переключатели (*switches*) 126—127
- Переменная (*variable*) 20—22
- Персональная машина последовательного вывода (*Personal Sequential Inference Machine*) 326
- Подстановка, дающая правильный ответ (*correct answer substitution*) 287
- Подстановочность (*substitutivity*) 301
- Подстановочный пример (*substitution instance*) 36
- Полная правильность (*total correctness*) 173, 177, 190
- Полное пространство вычислений (*total computation space*) 64
- Полнота (*completeness*)
 - множества процедур (*of procedure set*) 173
 - программы (*of program*) 165, 169, 172—173
 - резолюции (*of resolution*) 47, 287
- Порождение лемм (*lemma generation*) 154—158, 310
- Правила выбора (*selection rules*) 64—68
 - эквивалентной подстановки (*equivalence substitution rules*) 217—219
- Правило вывода (*inference rule*) 32
- вычислений (*computation rule*) 69, 184, 288
- поиска (*search rule*) 71—72
- Правильность алгоритмов (*correctness of algorithms*) 181—190
 - программ (*correctness of programs*) 166, 169—179
- Предикат (*predicate*) 19—20, 22
- Предикатный символ (*predicate symbol*) 19

- Предложение (*sentence*) 16, 22—23
- общезначимое (*valid sentence*) 48
- родительское (*parent sentence*) 32
- Представление данных (*data representation*) 245—252
- Преобразование определяющих (*definiens transformation*) 192
- Преобразование, улучшающее эффективность (*efficiency-improving transformation*) 114
- Предфиксная запись (*prefix notation*) 23
- Пример (*instance*) см. Подстановочный пример
- Примеры логических программ (*logic program examples*)
 - — — для выбора пар элементов (*for pair selection*) 208—213
 - — — — вычисления квадратного корня (*for square root estimation*) 85
 - — — — вычисления факториала (*for factorials*) 86, 89, 102, 104
 - — — — задачи о палиндроме (*for palindrome testing*) 139, 141, 149
 - — — — задачи о подмножествах (*for subset problem*) 168, 191
 - — — — линейного поиска в списке (*for linear search through lists*) 111, 112
 - — — — нахождения путей в графе (*for path finding*) 105, 106
 - — — — нахождения следующего элемента в списке (*for consecutivity testing in lists*) 39, 42, 62, 67, 138, 148
 - — — — нахождения собственного вектора матрицы (*for eigenproblems*) 107, 109, 161
 - — — — обнаружения пика (*for peak detection*) 112, 114, 126
 - — — — обращения списков (*for reversing lists*) 99, 100, 247
 - — — — подсчета числа различных элементов в списке (*for counting and filtering lists*) 96, 98
 - — — — поиска подстроки (*for substring searching*) 115, 117
 - — — — проверки упорядоченности списка (*for orderness testing in lists*) 214, 215, 222, 223
 - — — — распечатки списков (*for printing lists*) 155, 159
 - — — — склеивания списков (*for appending lists*) 133
 - — — — сложения векторов (*for*

- vector summation) 150, 151, 153, 155, 159
 — — — — — элементов списка (for addition of list members) 83
 — — — — удаления и вставки элемента в список (for deletion and insertion in lists) 119—121
Принцип рефлексии (reflection principle) 294
Присваивание данных (data assignment) 75, 239—243
Проблема общезначимости (validity problem) 48, 197
 — фреймов (frame problem) 296
Проблемная область (domain of problem) 16
Проверка вхождения (occur check) 272
Проект создания ЭВМ пятого поколения (Fifth Generation Project) 276, 301, 324—327
Производимость (producibility) 181—186
Противоречие (contradiction) 33—34
Протокол производителей-потребителей (producer-consumer protocol) 318
 — связанный (binding history) 43
Процедура (procedure) 53, 57—58
 — доказательства (proof procedure) 47
 — итеративная (iterative procedure) 83
 — рекурсивная (recursive procedure) 86

Разрешимая программа (solvable program) 67
Разрешимость (solvability) 179—181
Разыменование (dereferencing) 251
Раскрутка (bootstrapping) 270, 293
Распределение данных (data distribution) 75
Рассуждения на метауровне (meta-level reasoning) 292—298
Реализация Пролога на машине DEC-10 (DEC-10 PROLOG) 92, 255, 262, 270, 275
Резольвента (resolvent) 32, 286
Резолюция (resolution) 32, 47, 286—288
 — для вывода процедур (for procedure derivation) 208—209
 — как частичная разрешающая процедура (as partial decision procedure) 48
 — сверху вниз (top-down resolution) 34, 49
 — снизу вверх (bottom-up resolution) 49, 102—103
Рекурсия (recursion) 86—87, 99—101
 — хвостовая (tail recursion) 274
Реляционная модель баз данных (relational model of databases) 306—307
Референт (referent) 300
Референциальная прозрачность (referential transparency) 301
Решение задач (problem solving) 39—42, 49—50
 — пошаговое (incremental problem solving) 101—102

Сборка мусора (garbage collection) 258, 264
Свойство (property) 19
Связка (connective) 19
Связывание переменных (binding of variables) 43
Семантика (semantics)
 — логических программ (of logic programs) 281—286
 — неподвижной точки (fixpoint semantics) 281—286
 — непроцедурная (non-procedural semantics) 281—286
 — операционная (operational semantics) 57, 281
 — процедурная (procedural semantics) 57, 281
 — теоретико-модельная (model-theoretic semantics) 46, 282, 285
Семантическая сеть (semantic network) 313
Синтез программ (synthesis of programs) 198, 203—205
Система ALICE (ALICE system) 303, 323
 — APES (APES system) 311
 — CHAT-80 (CHAT-80 system) 308
Скелет (skeleton) 249
След (trail) 243—245
Совместное использование структур (structure sharing) 247—252, 266—269
Согласование параметров (parameter matching) 59, 124—125
Спецификация (specification) 165—166, 169—170
Специфицируемое отношение (specified relation) 166
 — решение (specified solution) 167
Список различий (difference list) 130

- Стандартная стратегия (standard strategy) 64, 68—74, 227—230
- Стек (stack)
- глобальный (global stack) 255—256, 266—267
 - исполнения (execution stack) 237, 239—243, 254—257
 - копий (copy stack) 266
 - локальный (local stack) 255—256, 266—267
- Стиль программирования (programming style) 73, 94
- Стратегия поиска в глубину (depth-first strategy) 69
- Структурная индукция (structural induction) 197
- Структурное программирование (structured programming) 94, 128, 223, 301
- Таблица истинности (truth table) 29
- решений (decision table) 13
- Тезис Черча (Church's Thesis) 279
- Тело процедуры (procedure body) 58
- Теорема Геделя о полноте (Godel Completeness Theorem) 285
- дедукции (Deduction Theorem) 207
 - о резолюции (Resolution Theorem) 41, 286
- Теорема Черча—Тьюринга (Church—Turing Theorem) 48
- Терм (term) 22
- как представление структуры данных (term-representation of data structures) 132, 135—143
 - простой тип данных (as simple data type) 135—137
- Точка возврата (backtrack point) 233
- Траектория управления (locus of control) 72, 228
- Указатель среды (environment pointer) 249
- Универсальная машина Тьюринга (universal Turing machine) 279
- Универсальность логики хорновских дизъюнкторов (universality of Horn clause logic) 278—281
- Унификатор (unifier) 36
- наиболее общий (most general unifier) 38
- Упорядочение (sequencing) 77—80
- Управление исполнением (control of execution) 227—239
- Управление на метауровне (metalevel control) 298
- Утверждение (statement) 51—53
- Факт (assertion) 32
- Фактический параметр (actual parameter) 57
- Формальный параметр (formal parameter) 58
- Формула (formula) 19—20, 22
- Фрейм (frame) 230—233, 235, 237, 253—256
- глобальный (global frame) 255
 - локальный (local frame) 255
 - родительский (parent frame) 231
 - удаляемый (deletable frame) 260
- Функтор (functor) 17
- Функциональное программирование (functional programming) 298—326
- Функциональный символ (functional symbol) 17
- Функция непосредственного следования (immediate consequence function) 285
- Хорновский дизъюнкт (Horn clause)
- 49
 - бинарный (binary Horn clause) 279
- Целевая переменная (goal variable) 55
- Целевое утверждение (goal) 53, 57
- — наиболее общее (most general goal) 167
- Частичная правильность (partial correctness) 169—173
- Частично рекурсивная функция (partial recursive function) 279
- Человеко-машинный интерфейс (man-machine interface) 304, 312
- Чистота интерпретатора (purity of interpreter) 92
- Шаг вывода (inference step) 32
- Эвристика (heuristic) 305, 310
- Экспертная система (expert system) 309—312
- Экстенсиональное представление (extensional representation) 144
- Эрбрановская интерпретация (Herbrand interpretation) 283
- модель (Herbrand model) 283—284
- Эрбрановский базис (Herbrand base) 290
- универсум (Herbrand universe) 278, 283

Оглавление

Предисловие редактора перевода	5
Предисловие	7
Предисловие автора	8
Введение	11
I. Представление знаний и рассуждения	16
I. 1. Индивидуумы	16
I. 2. Отношения	17
I. 3. Предикаты, связки и формулы	19
I. 4. Переменные	20
I. 5. Предложения	22
I. 6. Примеры представлений	23
I. 7. Интерпретация предложений	26
I. 8. Логическое следствие	30
I. 9. Логический вывод	32
I. 10. Общая резолюция сверху вниз	35
I. 11. Решение задач	39
I. 12. Извлечение ответа	42
I. 13. Резюме	45
I. 14. Исторический очерк	46
II. Логические программы	51
II. 1. Программы, вычисления и исполнение программ	51
II. 1.1. Утверждения в программах	51
II. 1.2. Назначение и структура программ	54
II. 1.3. Исполнение программ	55
II. 1.4. Вычисления	56
II. 2. Процедурная интерпретация	57
II. 2.1. Структура целевого утверждения	57
II. 2.2. Структура процедуры	57
II. 2.3. Операция вызова процедуры	58
II. 2.4. Вход в процедуру и выход из процедуры	60
II. 2.5. Выбор вызова	61
II. 2.6. Выбор процедуры	62
II. 3. Пространство вычислений	63
II. 3.1. Классификация вычислений	63
II. 3.2. Полное пространство вычислений	64
II. 3.3. Действие правил выбора	64
II. 4. Стандартная стратегия управления	68
II. 4.1. Недетерминированность и поиск	68
II. 4.2. Правило вычислений	69
II. 4.3. Правило поиска	71
II. 4.4. Стандартная стратегия	73

II. 5. Поведение программы в ходе вычислений	74
II. 5.1. Обработка данных	74
II. 5.2. Упорядочение	77
II. 5.3. Ветвление	80
II. 5.4. Итерация	82
II. 5.5. Рекурсия	86
II. 6. Встроенные средства	87
II. 6.1. Встроенные процедуры	88
II. 6.2. Встроенные функции	89
II. 7. Исторический очерк	90
III. Стиль программирования	94
III. 1. Логика и управление	95
III. 2. Итерация и рекурсия	99
III. 3. Поведение сверху вниз и снизу вверх	101
III. 3.1. Вычисления факториалов	102
III. 3.2. Нахождение путей в графах	104
III. 3.3. Задача нахождения собственных значений и собственных векторов матрицы	106
III. 4. Детерминизм и недетерминизм	110
III. 4.1. Поиск в списке	110
III. 4.2. Обнаружение пика	112
III. 4.3. Задача о подстроке	114
III. 5. Отрицание	118
III. 6. Согласование параметров	124
III. 7. Переключатели	126
III. 8. Исторический очерк	127
IV. Структуры данных	131
IV. 1. Представление и выборка данных	132
IV. 1.1. Термы и факты	132
IV. 1.2. Прямой и косвенный доступ	134
IV. 2. Представления посредством структурированных термов	135
IV. 2.1. Некоторые общие типы данных	135
IV. 2.2. Программы нахождения следующего элемента	137
IV. 2.3. Программы для задачи о палиндроме	139
IV. 2.4. Контроль соответствия типов	142
IV. 3. Представления посредством фактов	143
IV. 3.1. Общие принципы	144
IV. 3.2. Контроль соответствия типов	145
IV. 3.3. Индексирование	146
IV. 3.4. Массивы и индексы	148
IV. 4. Обработка данных в виде фактов	150
IV. 4.1. Имена как выходные данные	150
IV. 4.2. Факты как выходные данные: метод, используемый в Прологе	151
IV. 4.3. Факты как выходные данные: порождение лемм	154
IV. 4.4. Имена для структур данных, представленных в виде фактов	158
IV. 4.5. Еще раз о задаче нахождения собственных значений и собственных векторов	160
IV. 5. Исторический очерк	163
V. Верификация программ	165
V. 1. Вычисляемые и специфицируемые отношения	165
V. 1.1. Обозначения и терминология	166
V. 1.2. Вычисляемое отношение	168

V. 2. Правильность программ: определения	169
V. 2.1. Частичная правильность	170
V. 2.2. Полнота	172
V. 2.3. Полная правильность	173
V. 3. Правильность программ: достаточные условия	174
V. 4. Разрешимость	179
V. 5. Правильность алгоритмов: определения	181
V. 5.1. Производимость	181
V. 5.2. Сравнение трех алгоритмов	182
V. 5.3. Завершаемость	186
V. 5.4. Определения правильности логических алгоритмов	188
V. 6. Правильность алгоритмов: достаточные условия	189
V. 7. Пример верификации	190
V. 7.1. Выбор критериев	191
V. 7.2. Выбор спецификации	192
V. 7.3. Преобразование определяющих	192
V. 7.4. Выбор процедур	193
V. 7.5. Доказательство завершаемости	196
V. 8. Ограничения на верификацию	197
V. 9. Исторический очерк	198
VI. Формальный синтез программ	203
VI. 1. Правильность программ	203
VI. 2. Синтез логических программ	204
VI. 3. Синтез программ при помощи вывода процедур	205
VI. 4. Пример с использованием резолюции	208
VI. 5. Пример с использованием нерезолютивного вывода	213
VI. 5.1. Исходная программа	213
VI. 5.2. Изменение представления данных	214
VI. 5.3. Схема новой программы	215
VI. 5.4. Спецификация	217
VI. 5.5. Эквивалентная подстановка	217
VI. 5.6. Вывод новых процедур	219
VI. 5.7. Дальнейшие усовершенствования программы	221
VI. 6. Исторический очерк	223
VII. Реализация	226
VII. 1. Представление состояния управления	227
VII. 1.1. Механизм исполнения	227
VII. 1.2. Фреймы	230
VII. 1.3. Механизм возврата	233
VII. 1.4. Шаг вызова процедуры	235
VII. 1.5. Алгоритм управления	237
VII. 2. Представление присваиваний данных	239
VII. 2.1. Одностековое представление	239
VII. 2.2. След	243
VII. 2.3. Представление данных	245
VII. 2.4. Совместное использование структур	247
VII. 3. Экономия памяти	252
VII. 3.1. Восстановление пространства после успешного выхода из процедуры	252
VII. 3.2. Двухстековое представление	254
VII. 3.3. Механизм восстановления при успешном выходе из процедуры	257
VII. 3.4. Оптимизация последнего вызова	259

VII. 4. Экономия времени обработки данных	265
VII. 4.1. Системы без совместного использования структур	266
VII. 4.2. Компиляция	269
VII. 4.3. Унификация	270
VII. 5. Исторический очерк	273
VIII. Вклад логического программирования в теорию вычислений	277
VIII. 1. Теория вычислений	277
VIII. 1.1. Представимость и вычислимость	278
VIII. 1.2. Семантика	281
VIII. 1.3. Корректность и полнота стратегии исполнения	286
VIII. 1.4. Отрицание	288
VIII. 1.5. Рассуждения на метауровне	292
VIII. 2. Вычислительная практика	298
VIII. 2.1. Методология программирования	298
VIII. 2.2. Приложения	304
VIII. 2.3. Обучение	313
VIII. 3. Вычислительная техника	315
VIII. 3.1. Логика как неформальный язык программирования	316
VIII. 3.2. Проект создания ЭВМ пятого поколения	324
Литература	328
Предметный указатель	339

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присыпать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., 2, издательство «Мир».

Учебное издание

**Кристофер Джон Хоггер
ВВЕДЕНИЕ В ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

Заведующий редакцией чл.-корр. АН СССР В. И. Арнольд

Зам. зав. редакцией А. С. Попов

Ст. научный ред. А. А. Брянданская

Мл. научный ред. Н. С. Полякова

Художник В. Б. Прищепа

Художественный редактор В. И. Шаповалов

Технический редактор Е. С. Потапенкова

Корректор Т. М. Подгорная

ИБ № 6391

Сдано в набор 04.02.88. Подписано к печати 21.11.88. Формат 60×90 1/16. Бумага типографская № 1. Печать высокая. Гарнитура литературная. Объем 11,00 бум. л. Усл. печ. л. 22,00. Усл. кр.-отт. 22,00. Уч.-изд. л. 22,21. Изд. № 1/5535. Тираж 15 000 экз. Заказ 247. Цена 1 р. 90 к.

Издательство «Мир» В/О «Совэкспорткнига» Государственного комитета СССР по делам издательства, полиграфии и книжной торговли, 129820, ГСП, Москва, 1-й Рижский пер. 2.

Отпечатано с набора Ленинградской типографии № 2 — головного предприятия ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательства, полиграфии и книжной торговли, 198052, г. Ленинград, Л-52, Измайловский проспект, 29, в Ленинградской типографии № 4 ордена Трудового Красного Знамени Ленинградского объединения «Техническая книга» им. Евгении Соколовой Союзполиграфпрома при Государственном комитете СССР по делам издательства, полиграфии и книжной торговли, 191126, Ленинград, Социалистическая ул., 14.

**В издательстве «МИР»
готовится к печати**

Хампель Ф., Рончетти Э., Рауссеу П., Штаэль В. Робастность в статистике. Подход на основе функций влияния: Пер. с англ.— М.: Мир, 1989, 35 л., 3 р. 80 к., 10 000 экз.

Монография известных зарубежных специалистов (Швейцария, США, Нидерланды), посвященная важному разделу современной математической статистики. В ней использован подход на основе чувствительности функционалов к изменениям выборки. Многие классы оценок рассмотрены впервые. Приведены описания пакетов программ, имеются упражнения для практического применения, дана обширная библиография.

Для специалистов разных областей науки, использующих и разрабатывающих статистические методы, для аспирантов и студентов вузов.

Из отзыва д-ра физ.-мат. наук В. М. Золотарева: «Книга ... отвечает нашему желанию видеть теорию робастности освоенным и постоянно совершенствуемым инструментом в руках наших ученых, в том смысле, что знакомство с ней позволит широкому кругу специалистов-математиков и практиков получить нужные знания в достаточно компактном и вполне доступном для освоивания виде».

Уважаемый читатель!

Заблаговременно оформляйте заказы на интересующие Вас книги. Заказы принимаются в магазинах, торгующих научно-технической литературой.

**В издательстве «МИР»
готовится к печати**

Вирт Н. Алгоритмы и структуры данных: Пер. с англ.— М.: Мир, 1989, 20 л., 1 р. 60 к., 50 000 экз.

Новая книга известного швейцарского специалиста содержит изложение фундаментальных принципов построения эффективных и надежных программ. Она представляет собой существенно переработанное издание его книги «Алгоритмы + структуры данных = программы» (М.: Мир, 1985). В основу изложения положен язык Модула-2, который весьма распространен и как учебный, и как язык программирования управляющих систем. Многие примеры улучшены или переработаны. Ряд разделов расширен за счет описания новых принципов решения некоторых проблем.

Для программистов разной квалификации, преподавателей и студентов, специализирующихся по математическому обеспечению ЭВМ.

Книга рекомендована к переводу чл.-корр. АН СССР Г. Г. Рябовым и д-ром физ.-мат. наук Д. Б. Подшиваловым.

Уважаемый читатель!

Заблаговременно оформляйте заказы на интересующие Вас книги. Заказы принимаются в магазинах, торгующих научно-технической литературой.

ISBN 5-03-000490-4 (русск.)
ISBN 0-12-252090-8 (англ.)